Finally, to execute the program CLEAN.COM, type

```
-G  <Enter>
```

The result is the same as if the CLEAN.COM program had been run from the MS-DOS command level with the entry

```
C>CLEAN MYFILE.DAT  <Enter>
```

except that the program is executing under the control of DEBUG and within DEBUG's memory buffer.

DEBUG: O

Output to Port

Purpose

Writes 1 byte to an input/output (I/O) port.

Syntax

O *port byte*

where:

port	is an I/O port address from 0 through FFFFH.
byte	is a value from 0 through 0FFH to be written to the I/O port.

Description

The Output to Port (O) command writes 1 byte of data to the specified I/O port address. The data value must be in the range 00H through 0FFH.

Warning: The O command should be used with caution because it directly accesses the computer hardware and no error checking is performed. Attempts to write to some port addresses, such as those for ports connected to peripheral device controllers, timers, or the system's interrupt controller, may cause the system to crash or damage data stored on disk.

Example

To write the value C8H to I/O port 10AH, type

```
-O 10A C8  <Enter>
```

DEBUG: P

Proceed Through Loop or Subroutine

Purpose

Executes a loop, repeated string instruction, software interrupt, or subroutine call to completion.

Syntax

P [=*address*] [*number*]

where:

address	is the location of the first instruction to be executed.
number	is the number of instructions to execute.

Description

The Proceed Through Loop or Subroutine (P) command transfers control from DEBUG to the target program. The program executes without interruption until the loop, repeated string instruction, software interrupt, or subroutine call at *address* is completed or until the specified number of machine instructions have been executed. Control then returns to DEBUG, and the contents of the target program's registers and the status of the flags are displayed.

If the *address* parameter does not include an explicit segment, DEBUG uses the target program's CS register; if *address* is omitted entirely, execution begins at the address specified by the target's CS:IP registers. The *address* parameter must be preceded by an equal sign (=) to distinguish it from *number*.

If the instruction at *address* is not a loop, repeated string instruction, software interrupt, or subroutine call, the P command functions just like the Trace Program Execution (T) command. The optional *number* parameter specifies the number of instructions to be executed before control returns to DEBUG. If *number* is omitted, DEBUG executes only one instruction. After each instruction is executed, DEBUG displays the contents of the target program's registers, the status of the flags, and the next instruction to be executed.

Warning: The P command cannot be used to trace through ROM.

Example

Assume that the target program's location CS:143FH contains a CALL instruction. To execute the subroutine that is the destination of CALL and then return control to DEBUG, type

```
-P =143F  <Enter>
```

DEBUG: Q

Quit

Purpose

Ends a DEBUG session.

Syntax

Q

Description

The Quit (Q) command terminates the DEBUG program and returns control to MS-DOS or the command shell that invoked DEBUG. Any changes to a program or other file that were not saved on disk with the Write File or Sectors (W) command are lost.

Example

To exit DEBUG, type

```
-Q  <Enter>
```

DEBUG: R

Display or Modify Registers

Purpose

Displays the contents of one or all registers and the status of the CPU flags and allows them to be modified.

Syntax

R [*register*]

where:

register is the two-character name of an Intel 8086/8088 register from the following list:

```
AX BX CX DX SP BP SI DI
DS ES SS CS IP PC
```

or the character F, which specifies the CPU flags.

Description

The Display or Modify Registers (R) command displays the target program's register contents and the status of the CPU flags and allows them to be modified.

If R is entered without a *register* parameter, the contents of all registers and the status of the CPU flags are displayed, followed by a disassembly of the machine instruction currently pointed to by the target program's CS:IP registers.

If *register* is included in the R command line, the contents of the specified register are displayed; then DEBUG prompts with a colon character (:) for a new value. The value is entered by typing one to four hexadecimal digits and then pressing the Enter key. Pressing the Enter key without entering any values leaves the register contents unchanged.

Note: The register name PC is not fully supported in some versions of DEBUG, so the register name IP should be used instead.

Specifying the character F instead of a register name causes DEBUG to display the status of the program's CPU flags as two-character codes from the following list:

Flag Name	Value If Set (1)	Value If Clear (0)
Overflow	OV (Overflow)	NV (No Overflow)
Direction	DN (Down)	UP (Up)
Interrupt	EI (Enabled)	DI (Disabled)

(more)

Flag Name	Value If Set (1)	Value If Clear (0)
Sign	NG (Minus)	PL (Plus)
Zero	ZR (Zero)	NZ (Not Zero)
Aux Carry	AC (Aux Carry)	NA (No Aux Carry)
Parity	PE (Even)	PO (Odd)
Carry	CY (Carry)	NC (No Carry)

After displaying the flag values, DEBUG displays a hyphen (-) prompt on the same line. Any or all flags can then be altered by typing one or more codes (in any order and optionally separated by spaces) from the list above and pressing the Enter key. Pressing the Enter key without entering any codes leaves the status of the flags unchanged.

Examples

To display the contents of the target program's CPU registers and the status of the CPU flags, followed by the disassembled mnemonic for the next instruction to be executed (pointed to by CS:IP), type

```
-R  <Enter>
```

This produces a display in the following format:

```
AX=0000  BX=0000  CX=00A1  DX=0000  SP=FFFE  BP=0000 SI=0000  DI=0000
DS=19A5  ES=19A5  SS=19A5  CS=19A5  IP=0100   NV UP EI PL NZ NA PO NC
19A5:0100 BF8000        MOV   DI,0080
```

To display the value of the target program's BX register, type

```
-R BX  <Enter>
```

If BX contains 0200H, for example, DEBUG displays that value and then issues a prompt in the form of a colon:

```
BX 0200
:
```

The contents of BX can then be altered by typing a new value and pressing the Enter key or left unchanged by pressing the Enter key alone.

To set the direction and carry flags, first type

```
-R F  <Enter>
```

DEBUG displays the flag values, followed by a hyphen (-) prompt:

```
NV UP EI PL NZ NA PO NC  -
```

The direction and carry flags can then be set by entering

```
-DN CY  <Enter>
```

Messages

bf Error
Bad flag: An invalid code for a CPU flag was entered.

br Error
Bad register: An invalid register name was entered.

df Error
Double flag: Two values for the same CPU flag were entered in the same command.

DEBUG: S

Search Memory

Purpose

Searches memory for a pattern of 1 or more bytes.

Syntax

S *range list*

where:

range	specifies the starting and ending addresses or the starting address and length of the area to be searched.
list	is 1 or more consecutive byte values and/or a string to be searched for.

Description

The Search Memory (S) command searches a designated range of memory for a specified list of consecutive byte values and/or a text string. The starting address of each set of matching bytes is displayed. The contents of the searched area are not altered.

The *range* parameter specifies the starting and ending addresses or the starting address and length in bytes of the area to be searched. If a segment is not included in *range*, DEBUG uses DS. If a segment is specified for the starting address, DEBUG uses the same segment for the ending address. If a starting address and length in bytes is specified, the starting address plus the length minus 1 cannot exceed FFFFH.

The *list* parameter specifies one or more consecutive hexadecimal byte values and/or a string to be searched for, separated by spaces, commas, or tab characters. Strings must be enclosed within single or double quotation marks, and case is significant within a string.

Examples

To search for the string *Copyright* in the area of memory from DS:0000H through DS:1FFFH, type

```
-S 0 1FFF 'Copyright'  <Enter>
```

or

```
-S 0 L2000 "Copyright"  <Enter>
```

If matches are found, DEBUG displays the starting address of each:

```
20A8:0910
20A8:094F
20A8:097C
```

To search for the byte sequence *3BH 06H* in the area of memory from CS:0100H through CS:12A0H, type

```
-S CS:100 12A0 3B 06  <Enter>
```

or

```
-S CS:100 L11A1 3B 06  <Enter>
```

DEBUG: T

Trace Program Execution

Purpose

Executes one or more instructions, displaying the CPU status after each instruction.

Syntax

T [=*address*] [*number*]

where:

address	is the location of the first instruction to be executed.
number	is the number of machine instructions to be executed.

Description

The Trace Program Execution (T) command executes one or more instructions, starting at the specified address, and after each instruction displays the contents of the CPU registers, the status of the flags, and the instruction pointed to by CS:IP.

Warning: The T command should not be used to execute any instructions that change the contents of the Intel 8259 interrupt mask (ports 20H and 21H on the IBM PC and compatibles) or to trace calls made to MS-DOS through Interrupt 21H. The Go (G) command should be used instead.

The *address* parameter points to the first instruction to be executed. If *address* does not include a segment, DEBUG uses the target program's CS register; if *address* is omitted entirely, execution begins at the address specified by the target program's CS:IP registers. If *address* is included, it must be preceded by an equal sign (=) to distinguish it from *number*.

The *number* parameter specifies the hexadecimal number of instructions to be executed before the DEBUG prompt is redisplayed (default = 1). Pressing Ctrl-C or Ctrl-Break interrupts execution of a sequence of T instructions. Consecutive instructions can then be executed individually by entering T commands with no parameters. Pressing Ctrl-S suspends execution and pressing any key then resumes the trace.

Note: The T command can be used to trace through ROM.

Example

To execute one instruction at location CS:1A00H and then return control to DEBUG, displaying the contents of the CPU registers and the status of the flags, type

```
-T =1A00  <Enter>
```

DEBUG: U

Disassemble (Unassemble) Program

Purpose

Disassembles machine instructions into assembly-language mnemonics.

Syntax

U [*range*]

where:

range specifies the starting and ending addresses or the starting address and length of the machine code to be disassembled.

Description

The Disassemble (Unassemble) Program (U) command translates machine instructions into assembly-language mnemonics.

The *range* parameter specifies the starting and ending addresses or starting address and length in bytes of the machine instructions to be disassembled. If *range* does not specify a segment, DEBUG uses CS. Note that if the starting address does not fall on an 8086 instruction boundary, the disassembly will be incorrect.

If *range* does not include a length or ending address, 32 (20H) bytes of memory are disassembled beginning at the specified starting address. If *range* is omitted, 32 bytes of memory are disassembled, starting at the address following the last instruction disassembled by the previous U command. If a U command has not been used before and *range* is omitted, disassembly begins at the address specified by the target program's CS:IP registers.

Note: The actual number of bytes displayed may vary slightly from the amount specified in *range* or from the default of 32 bytes because the length of instructions may vary. Also, the U command does not understand instructions specific to the 80186, 80286, and 80386 microprocessors. It displays such instructions as DBs.

Successive 32-byte fragments of code can be disassembled by entering additional U commands without parameters.

Example

To disassemble 8 bytes of machine instructions starting at CS:0100H, type

```
-U 100 107  <Enter>
```

or

```
-U 100 L8  <Enter>
```

DEBUG: W

Write File or Sectors

Purpose

Writes a file or individual sectors to disk.

Syntax

W [*address*]

or

W *address drive start number*

where:

address	is the first memory location of the data to be written.
drive	is the number of the destination disk drive (0 = drive A, 1 = drive B, 2 = drive C, and so on).
start	is the number of the first logical sector to write (0–FFFFH).
number	is the number of consecutive sectors to be written (0–FFFFH).

Description

The Write File or Sectors (W) command transfers a file or individual sectors from memory to the disk.

When the W command is entered without parameters or with only an address, the number of bytes specified by the contents of registers BX:CX is written from memory into the file named in the most recently used Name File or Command-Tail Parameters (N) command or the first file specified in the DEBUG command line if the N command has not been used. Files with a .EXE or .HEX extension cannot be written with the DEBUG W command.

Note: If a Trace Program Execution (T), Go (G), or Proceed Through Loop or Subroutine (P) command has been used or the contents of the BX or CX registers have been changed, the contents of BX:CX must be restored before the W command is used.

When *address* is not included in the command line, the target program's CS:0100H is assumed.

The W command can also be used to bypass the MS-DOS file system and directly access logical sectors on the disk. The memory address (*address*), disk drive number (*drive*), starting logical sector number (*start*), and number of sectors to be written (*number*) must all be provided in the command line in hexadecimal format. The W command should not be used to write sectors on network drives.

Warning: Extreme caution must be used with the W command. The disk's file structure can easily be damaged if the wrong parameters are entered.

Example

Assume that the interactive Assemble Machine Instructions (A) command was used to create a program in DEBUG's memory buffer that is 32 (20H) bytes long, beginning at offset 0100H. This program can be written to the file QUICK.COM by using the DEBUG Name File or Command-Tail Parameters (N), Display or Modify Registers (R), and Write File or Sectors (W) commands sequentially. First, use the N command to specify the name of the file to be written:

```
-N QUICK.COM  <Enter>
```

Next, use the R command to set registers BX and CX to the length to be written. Register BX contains the upper, or most significant half, of the length, whereas register CX contains the lower, or least significant half. Type

```
-R CX  <Enter>
```

DEBUG displays the contents of register CX and prompts with a colon (:). Enter the length after the prompt:

```
:20  <Enter>
```

To use the R command again to set register BX to zero, type

```
-R BX  <Enter>
```

followed by

```
:0  <Enter>
```

Finally, to create the disk file QUICK.COM and write the program into it, type

```
-W  <Enter>
```

DEBUG responds:

```
Writing 0020 bytes
```

Messages

EXE and HEX files cannot be written
Files with a .EXE or .HEX extension cannot be written to disk with the W command.

Writing *nnnn* bytes
After a successful write operation, DEBUG displays in hexadecimal format the number of bytes written to disk.

SYMDEB

Symbolic Debugger

Purpose

The Symbolic Debugger (SYMDEB) allows a file to be loaded, examined, altered, and written back to disk. If the file contains a program, the program can be disassembled, modified, traced one instruction at a time, or executed at full speed with breakpoints. SYMDEB can also be used to read, modify, and write absolute disk sectors.

The SYMDEB utility is supplied with the Microsoft Macro Assembler (MASM) versions 4.0 and earlier. This documentation describes SYMDEB version 4.0.

Syntax

SYMDEB

or

SYMDEB [*options*] [*symfile* [*symfile*...]] [*filename* [*parameter*...]]

where:

symfile	is the name of a symbol file created with the MAPSYM utility (extension = .SYM).
filename	is the name of the binary or executable program file to be debugged.
parameter	is a command-line parameter required by the program being debugged.
options	is one or more of the following switches. Switches can be either uppercase or lowercase and can be preceded by a dash (-) instead of a forward slash (/).

/I	(IBM) specifies that the computer is IBM compatible.
/K	enables the interactive breakpoint key (Scroll Lock).
/N	enables the use of nonmaskable interrupt break systems on IBM-compatible computers (requires special hardware).
/S	enables the Screen Swap (\) command on IBM-compatible computers (the /I switch is also required).
/"commands"	specifies one or more SYMDEB commands, separated by semicolons and enclosed in quotation marks.

Description

The SYMDEB commands and capabilities are a superset of those in DEBUG. SYMDEB is also able to load and interpret special symbol files that correlate line numbers, symbols, and memory addresses. With the aid of such files, SYMDEB enables the user to specify

addresses with labels, variable names, and expressions, rather than only with absolute hexadecimal addresses. SYMDEB's command repertoire also includes I/O redirection commands, floating-point number entry and display commands, and source-code display capabilities that are not present in DEBUG.

The SYMDEB command line typically includes the *filename* parameter, which is the name of an executable program (with the extension .COM or .EXE) to be loaded into SYMDEB's memory buffer. Files with the extension .EXE are loaded in a manner compatible with the MS-DOS loader. Files with the extension .HEX are converted to binary images and loaded at the internally specified address. All other files are assumed to be direct memory images and are read directly into memory starting at offset 100H. If SYMDEB is entered by itself, no file information is read into memory. An appropriate program segment prefix (PSP) is synthesized at the head of SYMDEB's buffer for use by the target program; the PSP includes a command tail at offset 80H and default file control blocks (FCBs) at offsets 5CH and 6CH, constructed from the optional parameters following *filename*. If necessary, contents of the file are relocated so that the file is ready to execute.

The command line can also contain the names of one or more *symfiles*, symbol files that contain symbol and line-number information for the object modules that constitute the program being debugged. A symbol file is created with the MAPSYM utility from a map file produced by the Microsoft Object Linker (LINK). A symbol file always has the extension .SYM. *See* PROGRAMMING UTILITIES: MAPSYM; LINK.

The four command-line switches /I, /K, /N, and /S provide SYMDEB with information about the computer on which the utility is running. The /I switch is used when the computer is IBM compatible; this causes SYMDEB to take full advantage of special hardware features such as the 8259 Programmable Interrupt Controller or the memory-mapped video display. The /K switch enables the interactive breakpoint key (Scroll Lock), which can then be pressed at any time to interrupt a program that is being traced under the control of SYMDEB.

Note: The /K switch is not necessary on an IBM PC/AT, because the Sys Req key is always active as an interactive break key.

The /N switch enables the use of the nonmaskable interrupt as a breakpoint signal on IBM-compatible computers; this interrupt is triggered by hardware-assisted debugging packages such as Periscope and Atron Corporation's Software Probe. The /S switch enables the Screen Swap (\) command, which allows the output from the program being traced to be maintained and displayed on demand on a virtual screen separate from the SYMDEB commands and messages.

Note: The /I, /N, and /S switches are unnecessary on personal computers built by IBM Corporation; SYMDEB automatically enables the capabilities provided by those switches when SYMDEB finds the IBM copyright notice in the machine's ROM.

After SYMDEB and any files named in the command line are loaded, SYMDEB displays its special prompt character, a hyphen (-), and awaits a command. SYMDEB commands consist of one or two letters, usually followed by one or more parameters. SYMDEB treats

uppercase and lowercase characters equivalently except when they are contained in strings enclosed within single or double quotation marks. SYMDEB does not execute commands until the Enter key is pressed.

The SYMDEB commands discussed in this section are

Command	Action
A	Assemble machine instructions.
BC	Clear breakpoints.
BD	Disable breakpoints.
BE	Enable breakpoints.
BL	List breakpoints.
BP	Set breakpoints.
C	Compare memory areas.
D	Display memory.
DA	Display ASCII.
DB	Display bytes.
DD	Display doublewords.
DL	Display long reals.
DS	Display short reals.
DT	Display 10-byte reals.
DW	Display words.
E	Enter data.
EA	Enter ASCII string.
EB	Enter bytes.
ED	Enter doublewords.
EL	Enter long reals.
ES	Enter short reals.
ET	Enter 10-byte reals.
EW	Enter words.
F	Fill memory.
G	Go execute program.
H	Perform hexadecimal arithmetic.
I	Input from port.
K	Perform stack trace.
L	Load file or sectors.
M	Move (copy) data.
N	Name file or command-tail parameters.
O	Output to port.
P	Proceed through loop or subroutine.
Q	Quit debugger.
R	Display or modify registers.
S	Search memory.

(more)

Command	Action
S+	Enable source display mode.
S–	Disable source display mode.
S&	Enable source and machine code display mode.
T	Trace program execution.
U	Disassemble (unassemble) program.
V	View source code.
W	Write file or sectors.
X	Examine symbol map.
XO	Open symbol map.
Z	Set symbol value.
<	Redirect SYMDEB input.
>	Redirect SYMDEB output.
=	Redirect SYMDEB input and output.
{	Redirect target program input.
}	Redirect target program output.
~	Redirect target program input and output.
\	Swap screen.
.	Display source line.
?	Help or evaluate expression.
!	Escape to shell.
*	Enter comment.

One or more SYMDEB commands, separated by semicolons and enclosed in double quotation marks, can be included in the original SYMDEB command line in the form /"*commands*" (for example, /"r;d;q"). These commands, which must precede the filename of the program being debugged, are carried out immediately when SYMDEB is loaded. (This is a convenient way to invoke SYMDEB and execute a series of batch commands.)

The parameters for a SYMDEB command include symbols; line numbers; addresses; ranges; and 8-bit, 16-bit, 32-bit, or floating-point values, expressions, and lists. Multiple parameters can be separated by spaces, tabs, or commas.

A symbol is a name that represents a register, an absolute value, a segment address, or a segment offset. A symbol consists of one or more characters but always begins with a letter, an underscore (_), a question mark (?), an at sign (@), or a dollar sign ($). The names of the various 8086/8088/80286 registers and CPU flags are built into SYMDEB and can be used at any time. Other symbols can be used only when one or more symbol files have been loaded in conjunction with the program to be debugged.

Note: SYMDEB regards symbols whose spellings differ only in case as the same symbol. A unique symbol name that does not conflict with programming instructions, register names, or hexadecimal numbers should always be used.

In MASM programs, symbols must be declared PUBLIC in the source code in order to be accessible during debugging (except for segment and group names, which are PUBLIC by default). In programs compiled with the current versions of Microsoft C, FORTRAN,

and Pascal, all symbols are passed through for debugging if the proper compilation switch is used; however, familiarity with the compiler's particular naming conventions is necessary (for example, the Microsoft C Compiler adds an underscore character to the beginning of every symbol).

A line number is a combination of decimal numbers, filenames, and symbols that specifies a unique line of text in a program source file. Line numbers always start with a dot character (.) and take one of the following forms:

.[*filename*:]*linenumber*
.+*displacement*
.–*displacement*
.*symbol*[+*displacement*]
.*symbol*[–*displacement*]

The second and third variations specify a line relative to the current line number; the fourth and fifth specify a line number relative to a designated symbol. Line numbers can be used only with programs developed with compilers that generate line-number information. Programs developed with MASM or an incompatible compiler cannot generate line numbers.

An address identifies a unique location in memory. An address can be a simple offset or a complete address consisting of two 16-bit values in the form segment:offset. Each component can be a valid symbol (including CS, DS, ES, or SS, in the case of segments), a 16-bit hexadecimal number in the range 0 through FFFFH, or a symbol plus or minus a displacement. When the segment portion of an address is absent, the segment specified in the previous instance of the same command is used; if no segment was previously specified, SYMDEB uses DS unless an A, G, L, P, T, U, or W command is used, in which case SYMDEB uses CS.

A range specifies an area of memory or a number of data items and can be expressed as either two addresses or a starting address and a length. A length is represented by the letter L followed by a hexadecimal value in the range 0 through FFFFH. The meaning of the length varies with the SYMDEB command used: The length can signify a number of bytes, words, doublewords, real numbers, machine instructions, or source-code lines. If a command requires a range and the ending address is not supplied, SYMDEB usually assumes 128 bytes.

A value represents an integral number and is a combination of one or more digits. The default base for values is hexadecimal, except in the case of floating-point numbers, but other bases can be used by appending a radix character (Y for binary, O or Q for octal, T for decimal, H for hexadecimal) in either uppercase or lowercase. For example, the following values are equivalent:

0040	0100Q
0040H	0100O
0064t	1000000Y

Doubleword (32-bit) values are entered as two hexadecimal integers separated by a colon character (:). Real numbers are always entered in decimal radix, with or without a decimal point or exponent. Leading zeros can be omitted.

An expression is a combination of symbols, numeric constants, and operators that evaluates to an 8-, 16-, or 32-bit value. An expression can be used in place of a simple value in any command. Unary address operators use DS as the default segment for addresses. Expressions are evaluated in order of operator precedence; operators with equal precedence are evaluated from left to right. Parentheses can be used to override the normal operator precedence.

The available unary operators, listed in order of precedence from highest to lowest, are

Operator	Meaning
+	Unary plus
–	Unary minus
NOT	One's (bitwise) complement
SEG	Segment address of operand
OFF	Offset of operand
BY	Low-order byte from specified address
WO	Low-order word from specified address
DW	Doubleword from specified address
POI	Pointer from specified address (same as DW)
PORT	Byte input from specified port
WPORT	Word input from specified port

The available binary operators, listed in order of precedence from highest to lowest, are

Operator	Meaning
*	Multiplication
/	Integer division
MOD	Modulus
:	Segment override
+	Addition
–	Subtraction
AND	Bitwise Boolean AND
XOR	Bitwise Boolean Exclusive OR
OR	Bitwise Boolean Inclusive OR

A list is composed of one or more values, expressions, or strings, separated by spaces or commas. A string is one or more ASCII characters, enclosed within single or double quotation marks. Case is significant within a string. If the same type of quote character that is used to delimit the string occurs inside the string, the character must be doubled inside the string in order to be interpreted correctly (for example,"A ""quoted"" word").

In a few cases, SYMDEB displays a specific and informative error message in response to an invalid command. In general, though, SYMDEB responds in a generic fashion, pointing to the approximate location of the error with a caret character (^), followed by the word *Error*. For example:

```
-D CS:100,CS:80  <Enter>
               ^ Error
```

SYMDEB maintains a set of virtual CPU registers and flags for a program being debugged. These registers can be examined and modified with SYMDEB commands. When a program is first loaded for debugging, the virtual registers are initialized with the following values:

Register	.COM Program	.EXE Program
AX	Valid drive code	Valid drive code
BX	Upper half of program size	Upper half of program size
CX	Lower half of program size	Lower half of program size
DX	Zero	Zero
SI	Zero	Zero
DI	Zero	Zero
BP	Zero	Zero
SP	FFFEH or top of available memory minus 2	Size of stack segment
IP	100H	Offset of entry point within target program's code segment
CS	PSP	Base of target program's code segment
DS	PSP	PSP
ES	PSP	PSP
SS	PSP	Base of target program's stack segment

Note: SYMDEB checks the first three parameters in the command line. If the second and third parameters are filenames, SYMDEB checks any drive specifications with those filenames to verify that they designate valid drives. Register AX contains one of the following codes:

Code	Meaning
0000H	The drives specified with the second and third filenames are both valid, or only one filename was specified in the command line.
00FFH	The drive specified with the second filename is invalid.
FF00H	The drive specified with the third filename is invalid.
FFFFH	The drives specified with the second and third filenames are both invalid.

Before SYMDEB transfers control to the target program, it saves the actual CPU registers and then loads them with the current values of the virtual registers; conversely, when control reverts to SYMDEB from the target program, the returned register contents are stored back into the virtual register set for inspection and alteration by the SYMDEB user.

Examples

To prepare the program CLEAN.ASM for debugging with SYMDEB, declare all vital labels, procedures, and variable names in the source program PUBLIC. To assemble the program, type

```
C>MASM CLEAN;  <Enter>
```

This produces the relocatable object module CLEAN.OBJ. Then, to link the object module, type

```
C>LINK /MAP CLEAN;  <Enter>
```

This results in the executable program file CLEAN.EXE and the map file CLEAN.MAP.

Note: The /MAP switch must be used even if a map file is specified in the command line. Finally, to create the symbol information file required by SYMDEB, type

```
C>MAPSYM CLEAN  <Enter>
```

At this point, begin symbolic debugging by typing

```
C>SYMDEB CLEAN.SYM CLEAN.EXE  <Enter>
```

Any run-time command-line parameters required by the CLEAN program may be placed in the SYMDEB command line after the filename CLEAN.EXE.

To prepare the program SHELL.C for debugging with SYMDEB, first compile the program with the switches that disable optimization and cause line-number information to be written to the relocatable object module:

```
C>MSC /Zd /Od SHELL;  <Enter>
```

Next, to convert the object module to an executable program and create a map file with line-number information, type

```
C>LINK /MAP /LI SHELL;  <Enter>
```

To create the symbol information file required by SYMDEB for symbolic debugging, type

```
C>MAPSYM SHELL  <Enter>
```

To begin debugging, type

```
C>SYMDEB SHELL.SYM SHELL.EXE  <Enter>
```

To use the SYMDEB utility to inspect or modify memory or to read, modify, and write absolute disk sectors, type

```
C>SYMDEB  <Enter>
```

Message

File not found
The filename supplied as the first parameter in the SYMDEB command line cannot be found.

SYMDEB: A

Assemble Machine Instructions

Purpose

Allows entry of assembler mnemonics and translates them into executable machine code.

Syntax

A [*address*]

where:

address is the starting location for the assembled machine code.

Description

The Assemble Machine Instructions (A) command accepts assembly-language statements, rather than hexadecimal values, for the Intel 8086/8088, 80186, and 80286 (running in real mode) microprocessors and the Intel 8087 and 80287 math coprocessors and assembles each statement into executable machine language.

The *address* parameter specifies the location where entry of assembly-language mnemonics will begin. If *address* is omitted, SYMDEB uses the last address generated by the previous A command; if there was no previous A command, SYMDEB uses the current value of the target program's CS:IP registers.

After the user enters an A command, SYMDEB prompts for each assembly-language statement by displaying the address (a segment and an offset) in which the assembled code will be stored. When the user presses the Enter key, SYMDEB translates the assembly-language statement and stores each byte of the resulting machine instruction sequentially in memory (overwriting any existing information), beginning at the displayed address. SYMDEB then displays the address following the last byte of the machine instruction to prompt the user to enter the next assembled instruction. The user can terminate assembly mode by pressing the Enter key in response to the address prompt.

The assembly-language statements accepted by the SYMDEB A command have some slight syntactic differences and restrictions compared with the Microsoft Macro Assembler programming statements. These differences can be summarized as follows:

- All numbers are assumed to be hexadecimal integers unless otherwise specified with a radix character suffix.
- Segment overrides must be specified by preceding the entire instruction with CS:, DS:, ES:, or SS:.
- File control directives (NAME, PAGE, TITLE, and so forth), macro definitions, record structures, and conditional assembly directives are not supported by SYMDEB.

- When the data type (word or byte) is not implicit in the instruction, the type must be specified by preceding the operand with BYTE PTR (or BY), WORD PTR (or WO), DWORD PTR (or DW), QWORD PTR (or QW), or TBYTE PTR (or TB).
- In a string operation, the size of the string must be specified with a B (byte) or W (word) added to the string instruction mnemonic (for example, LODSB or LODSW).
- The DB and DW instructions accept a parameter of the type *list* and assemble byte and word values directly into memory.
- The WAIT or FWAIT opcodes for 8087/80287 assembler statements are not generated by the system and must be coded explicitly. (Note: 8087/80287 instructions can be assembled if the system is not equipped with a math coprocessor, but the system will crash if an attempt is made to execute them.)
- Addresses must be enclosed in square brackets to be differentiated from immediate operands.
- Repeat prefixes such as REP, REPZ, and REPNZ can be entered either alone on a line preceding the statement they affect or on the same line immediately preceding the statement.
- The assembler will generate the optimal form (SHORT, NEAR, or FAR) for jumps or calls, depending on the destination address, but these can be overridden if the operand is preceded with a NEAR (or NE) or FAR prefix.
- The mnemonic for a FAR RETURN is RETF.

Examples

To begin assembling code at address CS:0100H, type

```
-A 100  <Enter>
```

To assemble the instruction sequence

```
LODS WORD PTR [SI]
XCHG BX,AX
JMP [BX]
```

beginning at address CS:0100H, the following dialogue would take place:

```
-A 100  <Enter>
1983:0100  LODSW  <Enter>
1983:0101  XCHG BX,AX  <Enter>
1983:0103  JMP [BX]  <Enter>
1983:0105  <Enter>
```

To continue assembling at the last address generated by a previous A command (1983:0105H in the preceding example), type

```
-A  <Enter>
```

SYMDEB: BC

Clear Breakpoints

Purpose

Permanently removes sticky breakpoints.

Syntax

BC *

or

BC *list*

where:

*	represents all sticky breakpoints.
list	is one or more integers (sticky breakpoint numbers) in the range 0 through 9.

Description

The Clear Breakpoints (BC) command permanently clears the sticky breakpoints previously set with the Set Breakpoints (BP) command. A sticky breakpoint remains in memory throughout a SYMDEB session, unlike a breakpoint set with the Go (G) command, which remains in effect only while the G command executes.

If an asterisk character (*) follows the BC command, SYMDEB deletes all sticky breakpoints. If a *list* parameter containing one or more sticky breakpoint numbers in the range 0 through 9 follows the BC command, SYMDEB selectively deletes sticky breakpoints. Each sticky breakpoint is assigned a number when the breakpoint is created with the BP command. The List Breakpoints (BL) command can be used to display all current sticky breakpoint locations and numbers. Breakpoint numbers should be separated by spaces.

Sticky breakpoints can be temporarily disabled with the Disable Breakpoints (BD) command and subsequently re-enabled with the Enable Breakpoints (BE) command.

Examples

To clear sticky breakpoints 0, 4, and 8, type

```
-BC 0 4 8  <Enter>
```

To clear all sticky breakpoints, type

```
-BC *  <Enter>
```

Messages

Bad breakpoint number! (0–9)
A sticky breakpoint number in the command line was not an integer in the range 0 through 9.

Breakpoint list or '*' expected!
The BC command was entered without parameters.

SYMDEB: BD

Disable Breakpoints

Purpose

Temporarily disables sticky breakpoints.

Syntax

BD *

or

BD *list*

where:

*	represents all sticky breakpoints.
list	is one or more integers (sticky breakpoint numbers) in the range 0 through 9.

Description

The Disable Breakpoints (BD) command temporarily disables the sticky breakpoints previously set with the Set Breakpoints (BP) command. A sticky breakpoint remains in memory throughout a SYMDEB session, unlike a breakpoint set with the Go (G) command, which remains in effect only while the G command executes.

If an asterisk character (*) follows the BD command, SYMDEB disables all sticky breakpoints. If a *list* parameter containing one or more sticky breakpoint numbers in the range 0 through 9 follows the BD command, SYMDEB selectively disables sticky breakpoints. Each sticky breakpoint is assigned a number when the breakpoint is created with the BP command. The List Breakpoints (BL) command can be used to display all current sticky breakpoint locations and numbers. Breakpoint numbers should be separated by spaces.

Sticky breakpoints disabled with the BD command can be re-enabled with the Enable Breakpoints (BE) command. The Clear Breakpoints (BC) command can be used to permanently delete a sticky breakpoint.

Examples

To disable sticky breakpoints 0, 4, and 8, type

```
-BD 0 4 8  <Enter>
```

To disable all sticky breakpoints, type

```
-BD *  <Enter>
```

Messages

Bad breakpoint number! (0-9)
A sticky breakpoint number in the command line was not an integer in the range 0 through 9.

Breakpoint list or '*' expected!
The BD command was entered without parameters.

SYMDEB: BE

Enable Breakpoints

Purpose

Enables disabled sticky breakpoints.

Syntax

BE *

or

BE *list*

where:

*	represents all sticky breakpoints.
list	is one or more integers (sticky breakpoint numbers) in the range 0 through 9.

Description

The Enable Breakpoints (BE) command enables the sticky breakpoints disabled with the Disable Breakpoints (BD) command. A sticky breakpoint remains in memory throughout a SYMDEB session, unlike a breakpoint set with the Go (G) command, which remains in effect only while the G command executes.

If an asterisk (*) character follows the BE command, SYMDEB enables all sticky breakpoints. If a *list* parameter containing one or more sticky breakpoint numbers in the range 0 through 9 follows the BE command, SYMDEB selectively enables sticky breakpoints. Each sticky breakpoint is assigned a number when the breakpoint is created with the Set Breakpoints (BP) command. The List Breakpoints (BL) command can be used to display all current sticky breakpoint locations and numbers. Breakpoint numbers should be separated by spaces.

Examples

To enable sticky breakpoints 0, 4, and 8, type

```
-BE 0 4 8  <Enter>
```

To enable all sticky breakpoints, type

```
-BE *  <Enter>
```

Messages

Bad breakpoint number! (0-9)
A sticky breakpoint number in the command line was not an integer in the range 0 through 9.

Breakpoint list or '*' expected!
The BE command was entered without parameters.

SYMDEB: BL

List Breakpoints

Purpose

Displays information about all sticky breakpoints.

Syntax

BL

Description

The List Breakpoints (BL) command lists the current status of each sticky breakpoint created with the Set Breakpoints (BP) command. A sticky breakpoint remains in memory throughout a SYMDEB session, unlike a breakpoint set with the Go (G) command, which remains in effect only while the G command executes.

The BL command lists each sticky breakpoint number, its status code, its address in the target program, the number of passes remaining, and the initial number of passes specified with the BP command (in parentheses). If source display mode was selected with the Enable Source Display Mode (S+) command, SYMDEB also displays the source-file name and the line number that corresponds to each breakpoint location. Breakpoint status codes are

e Enabled
d Disabled
v Virtual

(A virtual breakpoint is a sticky breakpoint set at a symbol contained in a .EXE file that has not yet been loaded into SYMDEB.)

Example

To view the current status of all breakpoints, type

```
-BL  <Enter>
```

If the BP commands

```
-BP0 _TEXT:_main  <Enter>
-BP1 _TEXT:_printf  <Enter>
```

were previously entered, the BL command displays

```
0 e 456E:0010 [_TEXT:_main] dump.C:32
1 e 456E:0612 [_TEXT:_printf]
```

SYMDEB: BP

Set Breakpoints

Purpose

Sets sticky breakpoint locations within the program being debugged.

Syntax

BP[*n*] *address* [*passcount*] ["*commands*"]

where:

n	is the sticky breakpoint number (0–9).
address	is the location of the breakpoint in the target program.
passcount	is the number of times the instruction at *address* should be executed before the breakpoint is taken.
"*commands*"	is one or more SYMDEB commands, separated by semicolons. The entire list must be enclosed in double quotation marks. (Limit = 30 characters.)

Description

The Set Breakpoints (BP) command sets a sticky breakpoint in the program being debugged. A sticky breakpoint remains in memory throughout a SYMDEB session, unlike a breakpoint set with the Go (G) command, which remains in effect only while the G command executes. When the target program reaches the breakpoint, execution of the program is suspended and control returns to SYMDEB. SYMDEB displays the contents of the registers and flags, followed by a prompt so that the user can enter more commands.

The optional *n* parameter associates an integer in the range 0 through 9, called the breakpoint number, with the sticky breakpoint location. If *n* is omitted, the next available breakpoint number is used. No space is allowed between BP and *n*.

The *address* parameter must point to the first byte of a machine instruction in the program. This parameter may be a symbol, a literal address, or a source-code line number. If a segment is not included, SYMDEB uses the target program's CS register.

The optional *passcount* parameter is the number of times execution should pass through the specified location before the break is taken and control is returned to SYMDEB. The value of *passcount* must be a hexadecimal number in the range 0 through FFFFH (default = 0).

The optional "*commands*" parameter is one or more SYMDEB commands with their associated parameters. Each command must be separated from the others by a semicolon character (;) and the entire list enclosed in double quotation marks ("). A maximum of 30 characters can be specified within the quotation marks. The commands are executed whenever the break is taken.

Examples

To set a sticky breakpoint at location *next_file* in the target program and dump the contents of memory locations DS:0000H through DS:00FFH when the breakpoint is reached, type

```
-BP NEXT_FILE "DB DS:0 L100"  <Enter>
```

To associate the breakpoint number 4 with the location CS:4230H in the program being debugged and pass the breakpoint 16 (10H) times before suspending execution of the program, type

```
-BP4 CS:4230 10  <Enter>
```

Messages

Bad breakpoint number! (0–9)
A sticky breakpoint number in the command line was not an integer in the range 0 through 9.

Breakpoint command too long!
The "*commands*" parameter exceeded 30 characters.

Breakpoint error!
The BP command was entered without an *address* parameter.

Breakpoint redefined!
A new address was assigned to an existing breakpoint number, or an attempt was made to create a breakpoint with the same address as an existing breakpoint.

Duplicate breakpoint ignored!
An attempt was made to change an existing breakpoint to a breakpoint already specified in the breakpoint list.

Too many breakpoints!
No more sticky breakpoints are available.

SYMDEB: C

Compare Memory Areas

Purpose

Compares two areas of memory and reports any differences.

Syntax

C *range address*

where:

range	specifies the starting and ending addresses or the starting address and length of the first area of memory to be compared.
address	points to the beginning of the second area of memory to be compared.

Description

The Compare Memory Areas (C) command compares the contents of two areas of memory. The location and contents of any differing bytes are listed in the following form:

address1 byte1 byte2 address2

If no differences are found, the SYMDEB prompt returns.

The *range* parameter specifies the first through last addresses or the starting address and length in bytes of the first area of memory to be compared.

The *address* parameter points to the beginning of the second area of memory to be compared, which is the same size as *range*. If a segment is not included in either *range* or *address*, SYMDEB uses DS.

Example

To compare the 64 bytes beginning at CS:CE00H with the 64 bytes beginning at CS:CF0AH, type

```
-C CS:CE00,CE3F CS:CF0A  <Enter>
```

or

```
-C CS:CE00 L40 CS:CF0A  <Enter>
```

If any differences are found, SYMDEB displays them in the following format:

```
2124:CE06  00  FF  2124:CF10
```

SYMDEB: D

Display Memory

Purpose

Displays the contents of an area of memory.

Syntax

D [*range*]

where:

range specifies the starting and ending addresses or the starting address and length of the area of memory to be displayed.

Description

The Display Memory (D) command displays the contents of a specified range of memory addresses in the same format used in the most recent Display command (DA, DB, DD, DL, DS, DT, or DW). If no Display command has previously been entered, the memory is displayed in hexadecimal bytes and their ASCII equivalents (the DB format).

The *range* parameter specifies the starting and ending addresses of the memory area to be displayed or the starting address followed by the length of the area, expressed by an L and the hexadecimal number of data items to be displayed. When *range* does not include a segment, SYMDEB uses DS.

The size in bytes of each item and the default value for the length depend on the type of Display command used: the Display Byte (DB), Display Doubleword (DD), and Display Word (DW) commands default to a length of 128 (80H) bytes; Display ASCII (DA) displays 128 bytes or up to a null byte, whichever is smaller; Display Short Reals (DS), Display Long Reals (DL), and Display 10-Byte Reals (DT) default to the display of one floating-point number.

If a Display command has not previously been used and *range* is omitted from a D command, the display starts at the address specified in the target program's CS:IP registers. If a Display command has previously been used and *range* is omitted from a D command, the display starts at the memory address following the last address displayed by the most recent Display command.

Examples

Assume that the only Display commands used during this SYMDEB session are D and DB. To display the contents of the 128 bytes of memory beginning at offset 100H in the program's DGROUP, type

```
-D DGROUP:0100  <Enter>
```

SYMDEB displays the contents of the range of memory addresses in the following format:

```
7F00:0100  20 64 65 76 69 63 65 0D-0A 00 60 39 0D 0A 00 7C   device...'9...¦
7F00:0110  39 08 20 08 00 81 39 04-1B 5B 32 4A 42 BD 11 44  9. ...9..[2JB=.D
7F00:0120  2E 26 45 AF 11 47 B3 11-48 A5 11 4C B8 11 4E D3  .&E/.G3.H%.L8.NS
7F00:0130  11 50 DF 11 51 AB 11 54-DF 1E 56 37 11 5F 9F 16  .P_.Q+.T_.V7._..
7F00:0140  24 C0 11 00 03 4E 4F 54-C1 07 0A 45 52 52 4F 52  $@...NOTA..ERROR
7F00:0150  4C 45 56 45 4C 85 08 05-45 58 49 53 54 18 08 00  LEVEL...EXIST...
7F00:0160  03 44 49 52 03 91 0C 06-52 45 4E 41 4D 45 01 C0  .DIR....RENAME.@
7F00:0170  0F 03 52 45 4E 01 C0 0F-05 45 52 41 53 45 01 68  ..REN.@..ERASE.h
```

To view the next 128 bytes of memory, type

```
-D  <Enter>
```

SYMDEB displays the contents of memory addresses 7F00:0180H through 7F00:01FFH.

SYMDEB: DA

Display ASCII

Purpose

Displays the contents of memory in ASCII format.

Syntax

DA [*range*]

where:

range specifies the starting and ending addresses or the starting address and length of the area of memory to be displayed.

Description

The Display ASCII (DA) command displays the contents of a specified range of memory addresses in ASCII format.

The *range* parameter specifies the starting and ending addresses of the memory area to be displayed in ASCII format or the starting address followed by the length of the area, expressed by an L and a hexadecimal number of bytes. When *range* does not include a segment, SYMDEB uses DS.

If a Display command has not previously been used and *range* is omitted from a DA command, the display starts at the address specified in the target program's CS:IP registers. If a Display command has previously been used and *range* is omitted from a DA command, the display starts at the memory address following the last address displayed by the most recent Display command.

When a range is not explicit in a DA command, the display terminates after 128 bytes or when a null (zero) byte is encountered. If a range is specified, the entire range is displayed, including any null bytes, with nonprinting characters displayed as period (.) characters.

Each line of the display is formatted as a segment and offset, followed by the contents of 16 bytes of memory (or less if a null byte was encountered) represented as an ASCII string.

See also PROGRAMMING UTILITIES: SYMDEB:EA.

Examples

If memory beginning at location 7F00:0100H contains the characters *This is a test string* followed by a null (zero) byte, the command

```
-DA 7F00:0100  <Enter>
```

produces the following display:

```
7F00:0100  This is a test string
```

To view additional memory in the same format, type

```
-D  <Enter>
```

SYMDEB: DB

Display Bytes

Purpose

Displays the contents of memory as hexadecimal bytes and their equivalent ASCII characters.

Syntax

DB [*range*]

where:

range specifies the starting and ending addresses or the starting address and length of the area of memory to be displayed.

Description

The Display Bytes (DB) command displays the contents of a specified range of memory addresses as hexadecimal bytes and their ASCII character equivalents. This is the default format for the Display Memory (D) command.

The *range* parameter specifies the starting and ending addresses of the memory area to be displayed or the starting address followed by the length of the area, expressed by an L and a hexadecimal number of bytes. When *range* does not include a segment, SYMDEB uses DS.

If a Display command has not previously been used and *range* is omitted from a DB command, the display starts at the address specified in the target program's CS:IP registers. If a Display command has previously been used and *range* is omitted from a DB command, the display starts at the memory address following the last address displayed by the most recent Display command. When a range is not explicit in a DB command, the display terminates after 128 bytes.

Each line of the display is formatted as a segment and offset, followed by the contents of 16 bytes of memory represented as hexadecimal values separated by spaces (except the eighth and ninth values, which are separated by a dash), followed by their ASCII character equivalents (if any). In the ASCII section, nonprinting characters are displayed as periods.

See also PROGRAMMING UTILITIES: SYMDEB:EB.

Examples

To display the contents of the 128 bytes of memory beginning at 7F00:0100H, type

```
-DB 7F00:0100  <Enter>
```

The contents of the range of memory addresses are displayed in the following format:

```
7F00:0100  20 64 65 76 69 63 65 0D-0A 00 60 39 0D 0A 00 7C   device...'9...¦
7F00:0110  39 08 20 08 00 81 39 04-1B 5B 32 4A 42 BD 11 44  9. ...9..[2JB=.D
7F00:0120  2E 26 45 AF 11 47 B3 11-48 A5 11 4C B8 11 4E D3  .&E/.G3.H%.L8.NS
7F00:0130  11 50 DF 11 51 AB 11 54-DF 1E 56 37 11 5F 9F 16  .P_.Q+.T_.V7._..
7F00:0140  24 C0 11 00 03 4E 4F 54-C1 07 0A 45 52 52 4F 52  $@...NOTA..ERROR
7F00:0150  4C 45 56 45 4C 85 08 05-45 58 49 53 54 18 08 00  LEVEL...EXIST...
7F00:0160  03 44 49 52 03 91 0C 06-52 45 4E 41 4D 45 01 C0  .DIR....RENAME.@
7F00:0170  0F 03 52 45 4E 01 C0 0F-05 45 52 41 53 45 01 68  ..REN.@..ERASE.h
```

To view the next 128 bytes of memory, type

```
-D  <Enter>
```

SYMDEB displays the contents of memory addresses 7F00:0180H through 7F00:01FFH.

SYMDEB: DD

Display Doublewords

Purpose

Displays the contents of memory in hexadecimal doubleword format.

Syntax

DD [*range*]

where:

range specifies the starting and ending addresses or the starting address and length of the area of memory to be displayed.

Description

The Display Doublewords (DD) command displays the contents of a specified range of memory addresses 4 bytes at a time, as if they were FAR memory pointers (offset followed by segment in reverse byte order).

The *range* parameter specifies the starting and ending addresses of the memory to be displayed or the starting address followed by the length of the area, expressed by an L and a hexadecimal number of doublewords. When *range* does not include a segment, SYMDEB uses DS.

If a Display command has not previously been used and *range* is omitted from a DD command, the display starts at the address specified in the target program's CS:IP registers. If a Display command has previously been used and *range* is omitted from a DD command, the display starts at the memory address following the last address displayed by the most recent Display command. When a range is not explicit in a DD command, 32 doublewords (128 bytes) are displayed.

Each line of the display is formatted as a segment and offset, followed by the contents of 16 bytes of memory represented as 4 paired 16-bit segments and offsets. The 4 bytes that make up the segment and offset of each doubleword pointer are displayed in reverse order from their actual storage in memory.

See also PROGRAMMING UTILITIES: SYMDEB:ED.

Examples

To see how DD represents the 4 bytes that make up a doubleword, first type

```
-DB 100  <Enter>
```

This produces the following output:

```
3929:0100  CF 0B 9D 0D 33 0E C3 0E-F2 0E 06 0F 39 0F 49 0F 0...3.C.r...9.I.
```

Then type

```
-DD 100  <Enter>
```

This produces the following output:

```
3929:0100  0D9D:0BCF 0EC3:0E33 0F06:0EF2 0F49:0F39
```

Notice that DD switches the order of the first 2 bytes in a 4-byte set and designates them as the offset; then it switches the order of the second 2 bytes in the 4-byte set and designates them as the segment address.

To display the contents of the first 128 (80H) bytes of the system interrupt vector table, which is based at address 0000:0000H, type

```
-DD 0:0  <Enter>
```

This produces the following output:

```
0000:0000  2075:03D2 0070:01F0 16F3:2C1B 0070:01F0
0000:0010  0070:01F0 F000:FF54 F000:9805 F000:9805
0000:0020  0AE3:0395 16F3:2BAD F000:9805 F000:9805
0000:0030  0972:0B40 F000:9805 F000:EF57 0070:01F0
0000:0040  0AE3:03D6 F000:F84D F000:F841 0070:0D43
0000:0050  F000:E739 F000:F859 F000:E82E F000:EFD2
0000:0060  F000:E76C 0070:0ADD F000:FE6E 1078:3BEC
0000:0070  F000:FF53 F000:F0E4 0000:0522 F000:0000
```

To view the next 128 bytes of memory in the same format, type

```
-D  <Enter>
```

SYMDEB displays the contents of memory addresses 0000:0080H through 0000:00FFH.

SYMDEB: DL

Display Long Reals

Purpose

Displays the contents of memory as long (64-bit) floating-point numbers.

Syntax

DL [*range*]

where:

range specifies the starting and ending addresses or the starting address and length of the area of memory to be displayed.

Description

The Display Long Reals (DL) command displays the contents of a specified range of memory addresses 8 bytes at a time, as hexadecimal values and their decimal equivalents. The hexadecimal values are formatted as 64-bit floating-point numbers. The decimal values have the form

+¦−0.*decimaldigits*E+¦−*mantissa*

The sign of the number (+ or −) is followed by a zero, a decimal point, and a maximum of 16 *decimaldigits*; this, in turn, is followed by the designator of the mantissa (E) and the mantissa's sign (+ or −) and digits.

The *range* parameter specifies the starting and ending addresses of the memory to be displayed or the starting address followed by the length of the area, expressed by an L and a hexadecimal number of 8-byte values. When *range* does not include a segment, SYMDEB uses DS.

If a Display command has not previously been used and *range* is omitted from a DL command, the display starts at the address specified in the target program's CS:IP registers. If a Display command has previously been used and *range* is omitted from a DL command, the display starts at the memory address following the last address displayed by the most recent Display command. When a range is not explicit in a DL command, one 64-bit floating-point number is displayed.

Each line of the display is formatted as a segment and offset, followed by the contents of 8 bytes of memory represented as a hexadecimal value, followed by its decimal floating-point equivalent.

See also PROGRAMMING UTILITIES: SYMDEB:EL.

Examples

Assume that the memory beginning at location DS:0100H contains the value $6.624 * 10^{-27}$ (Planck's constant, in erg-seconds) as a 64-bit floating-point number. The command

```
-DL 100  <Enter>
```

produces the following output:

```
43E8:0100  5F A2 20 73 75 66 80 3A  +0.6624E-26
```

To view the next 8 bytes of memory in the same format, type

```
-D  <Enter>
```

SYMDEB: DS

Display Short Reals

Purpose

Displays the contents of memory as short (32-bit) floating-point numbers.

Syntax

DS [*range*]

where:

range specifies the starting and ending addresses or the starting address and length of the area of memory to be displayed.

Description

The Display Short Reals (DS) command displays the contents of a specified range of memory addresses 4 bytes at a time, as hexadecimal values and their decimal equivalents. The hexadecimal values are formatted as 32-bit floating-point numbers. The decimal values have the form

+¦−0.*decimaldigits*E+¦−*mantissa*

The sign of the number (+ or −) is followed by a zero, a decimal point, and a maximum of 16 *decimaldigits* (only the first 7 digits are significant); this, in turn, is followed by the designator of the mantissa (E) and the mantissa's sign (+ or −) and digits.

The *range* parameter specifies the starting and ending addresses of the area of memory to be displayed or the starting address followed by the length of the area, expressed by an L and a hexadecimal number of 4-byte values. When *range* does not include a segment, SYMDEB uses DS.

If a Display command has not previously been used and *range* is omitted from a DS command, the display starts at the address specified in the target program's CS:IP registers. If a Display command has previously been used and *range* is omitted from a DS command, the display starts at the memory address following the last address displayed by the most recent Display command. When a range is not explicit in a DS command, one 32-bit floating-point number is displayed.

Each line of the display is formatted as a segment and offset, followed by the contents of 4 bytes of memory represented as a hexadecimal value, followed by its decimal floating-point equivalent.

See also PROGRAMMING UTILITIES: SYMDEB:ES.

Examples

Assume that the memory beginning at location 43E8:0100H contains the value $6.02 * 10^{+23}$ (Avogadro's number) as a 32-bit floating-point number. The command

```
-DS 43E8:100  <Enter>
```

produces the following output:

```
43E8:0100  F9 F4 FE 66  +0.6020000172718952E+24
```

To view the next 4 bytes of memory in the same format, type

```
-D  <Enter>
```

SYMDEB: DT

Display 10-Byte Reals

Purpose

Displays the contents of memory as 10-byte (80-bit) floating-point numbers.

Syntax

DT [*range*]

where:

range specifies the starting and ending addresses or the starting address and length of the area of memory to be displayed.

Description

The Display 10-Byte Reals (DT) command displays the contents of a specified range of memory addresses 10 bytes at a time, as hexadecimal values and their decimal equivalents. The hexadecimal values are formatted as 80-bit floating-point numbers. (This format is ordinarily used by the Intel 8087 math coprocessor only for intermediate results during chained floating-point calculations.) The decimal value has the form

+¦–0.*decimaldigits*E+¦–*mantissa*

The sign of the number (+ or –) is followed by a zero, a decimal point, and a maximum of 16 *decimaldigits*; this, in turn, is followed by the designator of the mantissa (E) and the mantissa's sign (+ or –) and digits.

The *range* parameter specifies the starting and ending addresses of the area of memory to be displayed or the starting address followed by the length of the area, expressed by an L and a hexadecimal number of 10-byte values. When *range* does not include a segment, SYMDEB uses DS.

If a Display command has not previously been used and *range* is omitted from a DT command, the display starts at the address specified in the target program's CS:IP registers. If a Display command has previously been used and *range* is omitted from a DT command, the display starts at the memory address following the last address displayed by the most recent Display command. When a range is not explicit in a DT command, one 10-byte floating-point number is displayed.

Each line of the display is formatted as a segment and offset, followed by the contents of 10 bytes of memory represented as a hexadecimal value, followed by its decimal floating-point equivalent.

See also PROGRAMMING UTILITIES: SYMDEB:ET.

Examples

Assume that the memory beginning at location DS:0100H contains the value $2.99 * 10^{+10}$ (the speed of light in centimeters per second) as an 80-bit floating-point number. The command

```
-DT 100  <Enter>
```

produces the following output:

```
43E8:0100  00 00 00 00 60 B9 C5 DE 21 40  +0.299E+11
```

To view the next 10 bytes of memory in the same format, type

```
-D  <Enter>
```

SYMDEB: DW

Display Words

Purpose

Displays the contents of memory as 2-byte (16-bit) words.

Syntax

DW [*range*]

where:

range specifies the starting and ending addresses or the starting address and length of the area of memory to be displayed.

Description

The Display Word (DW) command displays the contents of a specified range of memory addresses 2 bytes at a time, as 16-bit hexadecimal integers.

The *range* parameter specifies the starting and ending addresses of the area of memory to be displayed or the starting address followed by the length of the area, expressed by an L and a hexadecimal number of words of memory to be displayed. When *range* does not include a segment, SYMDEB uses DS.

If a Display command has not previously been used and *range* is omitted from a DW command, the display starts at the address specified in the target program's CS:IP registers. If a Display command has previously been used and *range* is omitted from a DW command, the display starts at the memory address following the last address displayed by the most recent Display command. When a range is not explicit in a DW command, 64 words are displayed.

Each line of the display is formatted as a segment and offset, followed by the contents of 16 bytes of memory represented as eight 4-digit hexadecimal numbers. The 2 bytes that make up each word are displayed in reverse order from their actual storage in memory. That is, the first byte in a 2-byte word is displayed after the second byte.

See also PROGRAMMING UTILITIES: SYMDEB:EW.

Examples

To display the contents of the 64 words of memory beginning at DS:0080H in word format, type

```
-DW 80  <Enter>
```

This produces the following output:

```
1FEE:0080  6977 646E 776F 5C73 696C 0062 494C 3D42
1FEE:0090  3A63 6D5C 6373 6C5C 6269 633B 5C3A 6977
1FEE:00A0  646E 776F 5C73 696C 0062 4D54 3D50 3A63
1FEE:00B0  745C 6D65 0070 4554 504D 633D 5C3A 6574
1FEE:00C0  706D 4400 4149 3D4C 3A63 645C 6169 006C
1FEE:00D0  4350 3346 3D32 3A63 665C 726F 6874 705C
1FEE:00E0  3363 0032 4350 3350 3D32 3A63 665C 726F
1FEE:00F0  6874 705C 756C 3373 0032 5255 3146 3D30
```

To view the next 64 words of memory in the same format, type

```
-D  <Enter>
```

SYMDEB displays the contents of memory addresses 1FEE:0100H through 1FEE:017FH.

SYMDEB: E

Enter Data

Purpose

Enters data into memory.

Syntax

E *address* [*list*]

where:

address	is the first memory location for storage.
list	is the data to be placed into successive bytes of memory, starting at *address*.

Description

The Enter Data (E) command enters into memory one or more data items, using the same format as the most recent Enter command (EA, EB, ED, EL, ES, ET, or EW). If no Enter command has previously been used, the data can be entered as either hexadecimal values or ASCII strings (the EA or EB format). Any data previously stored at the specified locations is lost. If SYMDEB displays an error message, no changes are made.

The *address* parameter specifies the first byte to be modified. If *address* does not include a segment, SYMDEB uses DS. SYMDEB increments the address for each byte of data stored.

The *list* parameter must meet the requirements of the last Enter command used. All SYMDEB Enter commands are described in alphabetic order on the following pages. If *list* is included in the command line, the changes are made unless an error is detected in the command line. If *list* is omitted from the command line, the current contents of *address* are displayed, followed by a period (.), and the user is prompted for new data. If no value is entered and the Enter key is pressed, the original value remains unchanged and the Enter command is terminated.

Examples

The following two examples assume that no previous Enter commands have been used or that the most recent Enter command was EA or EB.

To store the byte values 00H, 0DH, and 0AH into the 3 bytes beginning at DS:1FB3H, type

```
-E 1FB3 00 0D 0A  <Enter>
```

If the command

```
-E 2C3 ABC  <Enter>
```

is entered and the last Enter command used was EA or EB, the value BCH is stored at DS:2C3H, and the leading 'A' character on the hexadecimal number 'ABC' is ignored.

SYMDEB: EA

Enter ASCII String

Purpose

Enters an ASCII string or hexadecimal byte values into memory.

Syntax

EA *address* [*list*]

where:

address	is the first memory location for storage.
list	is one or more ASCII strings or hexadecimal byte values.

Description

The Enter ASCII String (EA) command enters data into successive memory bytes. The data can be entered as either hexadecimal byte values or ASCII strings. Any data previously stored at the specified locations is lost. If SYMDEB displays an error message, no changes are made. The EA command functions exactly like the Enter Bytes (EB) command.

The *address* parameter specifies the first byte to be modified. If *address* does not include a segment, SYMDEB uses DS. SYMDEB increments the address for each byte of data stored.

The *list* parameter is one or more ASCII strings and/or hexadecimal byte values, separated by spaces, commas, or tab characters. Extra or trailing characters are ignored. Strings must be enclosed within single or double quotation marks, and case is significant within a string.

If *list* is included in the command line, the changes are made unless an error is detected in the command line. If *list* is omitted from the command line, the user is prompted byte by byte for new data, starting at *address*. The current contents of a byte are displayed, followed by a period. A new value for that byte can be entered as one or two hexadecimal digits (extra characters are ignored), or the contents can be left unchanged. To display the next byte, the user presses the spacebar. If the user enters a minus sign, or hyphen character (-), instead of pressing the spacebar, SYMDEB backs up to the previous byte. A maximum of 8 bytes can be entered on each input line; a new line is begun each time an 8-byte boundary is crossed. Data entry is terminated by pressing the Enter key without pressing the spacebar or entering any data.

Text strings can be used only as part of the *list* parameter in an EA command line; they cannot be entered in response to an address prompt.

Example

To store the string *MAIN MENU* into memory beginning at address ES:0C14H, type

```
-EA ES:C14 "MAIN MENU"  <Enter>
```

SYMDEB: EB

Enter Bytes

Purpose

Enters hexadecimal byte values or ASCII strings into memory.

Syntax

EB *address* [*list*]

where:

address	is the first memory location for storage.
list	is one or more hexadecimal byte values or ASCII strings.

Description

The Enter Bytes (EB) command enters data into successive memory bytes. The data can be entered as either hexadecimal byte values or ASCII strings. Any data previously stored at the specified locations is lost. If SYMDEB displays an error message, no changes are made. The EB command functions exactly like the Enter ASCII String (EA) command.

The *address* parameter specifies the first byte to be modified. If *address* does not include a segment, SYMDEB uses DS. SYMDEB increments the address for each byte of data stored.

The *list* parameter is one or more hexadecimal byte values and/or ASCII strings, separated by spaces, commas, or tab characters. Extra or trailing characters are ignored. Strings must be enclosed within single or double quotation marks, and case is significant within a string.

If *list* is included in the command line, the changes are made unless an error is detected in the command line. If *list* is omitted from the command line, the user is prompted byte by byte for new data, starting at *address*. The current contents of a byte are displayed, followed by a period. A new value for the byte can be entered as one or two hexadecimal digits (extra characters are ignored), or the contents can be left unchanged. To display the next byte, the user presses the spacebar. If the user enters a minus sign, or hyphen character (-), instead of pressing the spacebar, SYMDEB backs up to the previous byte. A maximum of 8 bytes can be entered on each input line; a new line is begun each time an 8-byte boundary is crossed. Data entry is terminated by pressing the Enter key without pressing the spacebar or entering any data.

Text strings can be used only as part of the *list* parameter in an EB command line; they cannot be entered in response to an address prompt.

Examples

To store the byte values 00H, 0DH, and 0AH into the 3 bytes beginning at DS:1FB3H, type

```
-EB 1FB3 00 0D 0A  <Enter>
```

To store the string *MAIN MENU* into memory beginning at address ES:0C14H, type

```
-EB ES:C14 "MAIN MENU"  <Enter>
```

SYMDEB: ED

Enter Doublewords

Purpose

Enters hexadecimal doubleword values into memory.

Syntax

ED *address*[*value*]

where:

address is the first memory location for storage.
value is a doubleword (32-bit) hexadecimal value.

Description

The Enter Doublewords (ED) command enters into memory 32-bit hexadecimal doubleword values in the form of FAR memory pointers (offset followed by segments in reverse byte order). Any data previously stored at the specified locations is lost. If SYMDEB displays an error message, no changes are made.

The *address* parameter specifies the first memory location to be modified. If *address* does not include a segment, SYMDEB uses DS.

The *value* parameter is one doubleword value, entered as two 16-bit hexadecimal words separated by a colon character (:). Each value is entered in the form segment:offset. The offset portion is stored at *address*, and the segment portion is stored at *address+2*, both in reverse byte order. For example, a value of AABB:CCDDH would be stored in memory as DDH, CCH, BBH, and AAH, starting at *address*. Multiple values cannot be used in an ED command line; SYMDEB ignores any values after the first value.

If *value* is omitted from the command line, SYMDEB prompts the user for new data, starting at *address*. The current contents of the location are displayed, followed by a period. The user can then enter a new doubleword value and press the Enter key or leave the contents unchanged by pressing the Enter key alone, which also terminates the ED command. If a new value is entered, SYMDEB increments *address* and displays the next doubleword value.

Example

To store the doubleword value F000:1392H at the address DS:0200H, type

```
-ED 200 F000:1392  <Enter>
```

SYMDEB: EL

Enter Long Reals

Purpose

Enters 64-bit floating-point numbers into memory.

Syntax

EL *address*[*value*]

where:

address	is the first memory location for storage.
value	is a 64-bit floating-point decimal number.

Description

The Enter Long Reals (EL) command enters into memory 64-bit floating-point numbers in decimal format. Any data previously stored at the specified memory locations is lost. If SYMDEB displays an error message, no changes are made.

The *address* parameter specifies the first byte to be modified. If *address* does not include a segment, SYMDEB uses DS.

The *value* parameter is a floating-point number entered in decimal radix, with or without a decimal point and/or exponent. Multiple values cannot be used in an EL command line; SYMDEB ignores any values after the first value.

The 64-bit floating-point decimal value must be entered in the form

[+¦–]*decimaldigits*[E[+¦–]*mantissa*]

where:

+¦–	is the sign of the long floating-point value or the mantissa.
decimaldigits	is a decimal number. A maximum of 16 digits is allowed, including digits before and after a decimal point.
E	denotes the beginning of the mantissa.
mantissa	is the decimal mantissa value.

If *value* is omitted from the command line, SYMDEB prompts the user for new data, starting at *address*. The current contents of the location are displayed. The user can enter a new value and press the Enter key or leave the contents unchanged by pressing the Enter key alone, which also terminates the EL command. If a new value is entered and the Enter key is pressed, SYMDEB increments *address* and displays the next long real number.

Example

To store an approximation of the value *pi* (π) in the form of a 64-bit floating-point number at address DS:0020H, type

```
-EL 20 +0.3141592653589793E+1  <Enter>
```

or

```
-EL 20 3.141592653589793  <Enter>
```

SYMDEB: ES

Enter Short Reals

Purpose

Enters 32-bit floating-point numbers into memory.

Syntax

ES *address* [*value*]

where:

address	is the first memory location for storage.
value	is a 32-bit floating-point decimal number.

Description

The Enter Short Reals (ES) command enters into memory 32-bit floating-point numbers in decimal format. Any data previously stored at the specified locations is lost. If SYMDEB displays an error message, no changes are made.

The *address* parameter specifies the first byte to be modified. If *address* does not include a segment, SYMDEB uses DS.

The *value* parameter is a floating-point number entered in decimal radix, with or without a decimal point and/or exponent. Multiple values cannot be used in an ES command line; SYMDEB ignores any values after the first value.

The 32-bit floating-point decimal value must be entered in the form

[+¦–]*decimaldigits*[E[+¦–]*mantissa*]

where:

+¦–	is the sign of the short floating-point value or the mantissa.
decimaldigits	is a decimal number. A maximum of 16 digits is allowed, including digits before and after a decimal point.
E	denotes the beginning of the mantissa.
mantissa	is the decimal mantissa value.

Note: For short floating-point values, the last nine *decimaldigits* are not significant. This can be demonstrated by using the Display Short Reals (DS) command to check the new value in memory.

If *value* is omitted from the command line, SYMDEB prompts the user for new data, starting at *address*. The current contents of the location are displayed. The user can then enter a new value and press the Enter key or leave the contents unchanged by pressing the

Enter key alone, which also terminates the ES command. If a new value is entered and the Enter key is pressed, SYMDEB increments *address* and displays the next short floating-point number.

Example

To store an approximation of the value *pi* (π) in the form of a 32-bit floating-point number at address DS:0020H, type

```
-ES 20 +0.31415927E+1  <Enter>
```

or

```
-ES 20 3.1415927  <Enter>
```

SYMDEB: ET

Enter 10-Byte Reals

Purpose

Enters 10-byte (80-bit) floating-point numbers into memory.

Syntax

ET *address* [*value*]

where:

address	is the first memory location for storage.
value	is an 80-bit floating-point decimal number.

Description

The Enter 10-Byte Reals (ET) command enters into memory 10-byte (80-bit) floating-point numbers in decimal format. Any data previously stored at the specified locations is lost. If SYMDEB displays an error message, no changes are made. (This 10-byte format is ordinarily used by the Intel 8087 math coprocessor only for intermediate results during chained floating-point calculations.)

The *address* parameter specifies the first memory location to be modified. If *address* does not include a segment, SYMDEB uses DS.

The *value* parameter is a floating-point number entered in decimal radix, with or without a decimal point and/or exponent. Multiple values cannot be used in an ET command line; SYMDEB ignores any values after the first value.

The 10-byte floating-point decimal value must be entered in the form

[+¦–]*decimaldigits*[E[+¦–]*mantissa*]

where:

+¦–	is the sign of the 10-byte floating-point value or the mantissa.
decimaldigits	is a decimal number. A maximum of 16 digits is allowed, including digits before and after a decimal point.
E	denotes the beginning of the mantissa.
mantissa	is the decimal mantissa value.

If *value* is omitted from the command, SYMDEB prompts the user for new data, starting at *address*. The current contents are displayed. The user can enter a new value and press the Enter key or leave the contents unchanged by pressing the Enter key alone, which also terminates the ET command. If a new value is entered and the Enter key is pressed, SYMDEB increments *address* and displays the next 10-byte floating-point number.

Example

To store an approximation of the value *pi* (π) in the form of an 80-bit floating-point number at address DS:0020H, type

```
-ET 20 +0.31415926535897932384E+1  <Enter>
```

or

```
-ET 20 3.1415926535897932384  <Enter>
```

SYMDEB: EW

Enter Words

Purpose

Enters word values into memory.

Syntax

EW *address* [*value*]

where:

address	is the first memory location for storage.
value	is a word (16-bit) hexadecimal value.

Description

The Enter Words (EW) command enters into memory 16-bit hexadecimal word values. Any data previously stored at the specified locations is lost. If SYMDEB displays an error message, no changes are made.

The *address* parameter specifies the first memory location to be modified. If *address* does not include a segment, SYMDEB uses DS.

The *value* parameter is one word value in the range 0 through FFFFH. The value is stored in reverse byte order. For example, a value of AABBH would be stored in memory as BBH and AAH, starting at *address*. Multiple values cannot be used in an EW command line; SYMDEB ignores any values after the first value.

If *value* is omitted from the command line, SYMDEB prompts the user word by word for new data, starting at *address*. The current contents are displayed, followed by a period. The user can enter a new word value as one to four hexadecimal digits and press the Enter key or leave the contents unchanged by pressing the Enter key alone, which also terminates the EW command. If a new value is entered, SYMDEB increments *address* and displays the next word value.

Example

To store the word value 1355H at the address DS:1C00H, type

```
-EW 1C00 1355 <Enter>
```

SYMDEB: F

Fill Memory

Purpose

Stores a repetitive data pattern into an area of memory.

Syntax

F *range list*

where:

range	specifies the starting and ending addresses or the starting address and length of memory to be filled.
list	is the data to be used to fill memory.

Description

The Fill Memory (F) command fills an area of memory with the data from a list. The data can be entered in either hexadecimal or ASCII format. Any data previously stored at the specified locations is lost. If SYMDEB displays an error message, no changes are made.

The *range* parameter specifies the starting and ending addresses or the starting address and hexadecimal length in bytes of the area of memory to be filled. If *range* does not include an explicit segment, SYMDEB uses DS.

The *list* parameter is one or more hexadecimal byte values and/or strings, separated by spaces, commas, or tab characters. Strings must be enclosed in single or double quotation marks, and case is significant within a string.

If the area to be filled is larger than the data list, the list is repeated as often as necessary to fill the area. If the data list is longer than the area of memory to be filled, the list is truncated to fit.

Examples

To fill the area of memory from DS:0B10H through DS:0B4FH with the value 0E8H, type

```
-F B10 B4F E8  <Enter>
```

or

```
-F B10 L40 E8  <Enter>
```

To fill the 16 bytes of memory beginning at address CS:1FA0H by replicating the 2-byte sequence 0DH 0AH, type

```
-F CS:1FA0 1FAF 0D 0A  <Enter>
```

or

```
-F CS:1FA0 L10 0D 0A  <Enter>
```

To fill the area of memory from ES:0B00H through ES:0BFFH by replicating the text string *BUFFER*, type

```
-F ES:B00 BFF "BUFFER"  <Enter>
```

or

```
-F ES:B00 L100 "BUFFER"  <Enter>
```

SYMDEB: G

Go

Purpose

Transfers execution control from SYMDEB to the target program being debugged.

Syntax

G[=*address*] [*break0*[... *break9*]]

where:

address	is the location at which to begin execution.
break0 ... *break9*	specify from 1 to 10 breakpoints.

Description

The Go (G) command transfers control from SYMDEB to the target program. If no breakpoints are set, the program will execute until it crashes or until it reaches a normal termination, in which case the message *Program terminated normally* is displayed and control returns to SYMDEB. (After this message has been displayed, it may be necessary to reload the program before it can be executed again.)

The *address* parameter can be any location in memory. If no segment is specified, SYMDEB uses the target program's CS register. If *address* is omitted, SYMDEB transfers to the current address in the target program's CS:IP registers. An equal sign (=) must precede *address* to distinguish it from the breakpoints *break0* ... *break9*.

The parameters *break0* ... *break9* specify from 1 to 10 breakpoints that can be set as part of the G command. Breakpoints can be placed in any order, because execution stops at the first breakpoint address encountered, regardless of the position of that breakpoint in the list. Each of the breakpoint addresses must contain the first byte of an 8086 opcode. SYMDEB installs breakpoints by replacing the first byte of the machine instruction at each breakpoint address with an Interrupt 03H instruction (opcode 0CCH). If the program encounters a breakpoint, program execution is suspended and control returns to SYMDEB. SYMDEB then restores the original machine code in the breakpoint locations, displays the contents of the current registers and flags and the instruction pointed to by CS:IP, and issues the standard SYMDEB prompt. If the target program executes to completion and terminates without encountering any of the breakpoints or is halted by some means other than a breakpoint, the Interrupt 03H instructions are not replaced with the original machine code and the Load File or Sectors (L) command must be used to reload the original program.

The G command requires that the target program's SS:SP registers point to a valid stack that has at least 6 bytes of stack space available. When the G command is executed, it

pushes the target program's flags and CS and IP registers onto the stack and then transfers control to the program with an IRET instruction. Thus, if the target program's stack is not valid or is too small, the system may crash.

The G command also recognizes any sticky breakpoints set with the Set Breakpoint (BP) command. These sticky breakpoints are not counted as part of the transient breakpoints specified in the G command line and are not removed after a breakpoint has been encountered.

Examples

To begin execution of the program in SYMDEB's buffer at location CS:110AH, setting breakpoints at CS:12FCH and CS:1303H, type

```
-G =110A 12FC 1303  <Enter>
```

To resume execution of the program following a breakpoint, type

```
-G  <Enter>
```

To begin execution at the label *main,* setting breakpoints at the procedures *fopen()* and *printf(),* type

```
-G =_main _fopen _printf  <Enter>
```

Messages

Program terminated normally
The program being debugged executed successfully without encountering any breakpoints and performed a normal termination with Interrupt 20H, Interrupt 21H Function 00H, or Interrupt 21H Function 4CH. If any breakpoints were set, the original program should be reloaded with the Load File or Sectors (L) command.

Too many breakpoints!
More than 10 breakpoints were specified in a Go (G) command. Enter the command again with 10 or fewer breakpoints.

SYMDEB: H

Perform Hexadecimal Arithmetic

Purpose

Displays the sum and difference of two hexadecimal numbers.

Syntax

H *value1 value2*

where:

value1 and *value2* are any two hexadecimal numbers in the range 0 through FFFFH.

Description

The Perform Hexadecimal Arithmetic (H) command displays the sum and difference of two 16-bit hexadecimal numbers—that is, the result of the operations *value1+value2* and *value1–value2*. If *value2* is greater than *value1*, SYMDEB displays their difference as a two's complement hexadecimal number. This command is convenient for performing quick calculations of addresses and other values during an interactive debugging session.

Examples

To display the sum and difference of the values 4B03H and 104H, type

```
-H 4B03 104  <Enter>
```

This produces the following display:

```
4C07  49FF
```

If the addition produces an overflow, the four least significant digits are displayed. For example, the command line

```
-H FFFF 2  <Enter>
```

produces the following display:

```
0001 FFFD
```

If *value2* is greater than *value1*, the difference is displayed in two's complement form. For example, the command line

```
-H 1 2  <Enter>
```

produces the following display:

```
0003 FFFF
```

SYMDEB: I

Input from Port

Purpose

Reads and displays 1 byte from an input/output (I/O) port.

Syntax

I *port*

where:

port is a 16-bit I/O port address in the range 0 through FFFFH.

Description

The Input from Port (I) command performs a read operation on the specified I/O port address and displays the data as a two-digit hexadecimal number.

Warning: This command must be used with caution because it involves direct access to the computer hardware and no error checking is performed. Input operations directed to the ports assigned to some peripheral device controllers may interfere with the proper operation of the system. If no device has been assigned to the specified I/O port or if the port is write-only, the value that will be displayed by an I command is unpredictable.

Example

To read and display the contents of I/O port 10AH, type

```
-I 10A  <Enter>
```

An example of the result of this command is

```
FF
```

SYMDEB: K

Perform Stack Trace

Purpose

Displays the current stack frame.

Syntax

K [*number*]

where:

number is the number of parameters supplied to the current procedure.

Description

The Perform Stack Trace (K) command displays the contents of the current stack frame. The first line of the display shows the name of the current procedure, parameters to the procedure, and the filename and line number of the call to the procedure. The subsequent lines trace the flow of execution that led to the current procedure.

In cases where SYMDEB cannot determine the number of parameters for a procedure by inspection of the stack frame (for example, if the number of parameters sent to a procedure varies), the *number* option can be used in the command to force the display of one or more parameters.

The K command can be used only on procedures that follow the calling conventions used by Microsoft high-level-language compilers.

Examples

Assume that a breakpoint has been set within the C library *printf()* routine, that the breakpoint has been reached, and that the SYMDEB prompt has reappeared. The command

```
-K  <Enter>
```

produces the following output:

```
_TEXT:_printf(00D4,0000,0000) from .dump.C:108
_TEXT:_dump_para(0000,0000,0FB8) from .dump.C:92
_TEXT:_dump_rec(0FB8,0001,0000,0000) from .dump.C:61
_TEXT:_main(?)
```

In this example, the breakpointed procedure *printf()* was called by the routine *dump_para()* with three parameters. *Dump_para()* was called by *dump_rec()*, which in turn was called by *main()*. Because SYMDEB cannot determine the depth of the stack

frame for the routine *main()*, it displays no parameters for it. The display of at least two parameters for every procedure can be forced by the command

```
-K 2  <Enter>
```

which produces the following example display:

```
_TEXT:_printf(00D4,0000,0000) from .dump.C:108
_TEXT:_dump_para(0000,0000,0FB8) from .dump.C:92
_TEXT:_dump_rec(0FB8,0001,0000,0000) from .dump.C:61
_TEXT:_main(0002,1044)
```

From a knowledge of C conventions, it follows that the first parameter for *main()* is *argc*, or the number of tokens in the command line that invoked the program being debugged; the second parameter is the offset within DGROUP of *argv*, or an array of pointers to each token.

SYMDEB: L

Load File or Sectors

Purpose

Loads a file or individual sectors from a disk.

Syntax

L [*address*]

or

L *address drive start number*

where:

address	is the starting address in memory that data read from a disk is placed into.
drive	is the decimal number (0-3) of the disk to read (0 = drive A, 1 = drive B, 2 = drive C, 3= drive D).
start	is the hexadecimal number of the first sector to load (0–FFFFH).
number	is the hexadecimal number of consecutive sectors to load (0–FFFFH).

Description

The Load File or Sectors (L) command loads a file or individual sectors from a disk.

When the L command is entered without parameters or with an address alone, the file specified in the SYMDEB command line or with the most recent Name File or Command-Tail Parameters (N) command is loaded from the disk into memory. If no segment is specified in *address*, SYMDEB uses CS. If the file's extension is .EXE, the file is placed in SYMDEB's target program buffer at the load address specified in the .EXE file's header; if the file's extension is .COM, the file is loaded at offset 100H. (If for some reason an address is entered for a .EXE or .COM file and the address is anything but 100H, an error message is displayed; if the address is 100H, it will be ignored.) If the file has a .HEX extension, the .HEX file's starting address is added to *address* before loading the file. If *address* is not specified, the .HEX file is placed at its own starting address. The length of the file or, in the case of a .EXE file, the actual length of the program (the length of the file minus the header) is placed in the target program's BX and CX registers, with the most significant 16 bits in register BX.

The L command can also be used to bypass the MS-DOS file system and obtain direct access to logical sectors on the disk. The memory address (*address*), disk drive number (*drive*), starting logical sector number (*start*), and number of sectors to read (*number*) must all be specified in the command line.

Note: The L command should not be used to access logical sectors on network drives.

Examples

To load the file specified in the SYMDEB command line or in the most recent N command into SYMDEB's target program buffer, type

```
-L  <Enter>
```

To load eight sectors from drive B, starting at logical sector 0, to memory location CS:0100H in SYMDEB's memory buffer, type

```
-L 100 1 0 8  <Enter>
```

Messages

Disk error reading disk *X*
A hardware-related disk error, such as a checksum error or seek incomplete, was encountered during the execution of an L command.

File not found
The file specified in the most recent N command cannot be found.

SYMDEB: M

Move (Copy) Data

Purpose

Copies the contents of one area of memory to another.

Syntax

M *range address*

where:

range	specifies the starting and ending addresses or the starting address and length of the area of memory to be copied.
address	is the first byte of the destination of the copy operation.

Description

The Move (Copy) Data (M) command copies data from one location in memory to another without altering the data in the original location. If the source and destination areas overlap, the data is copied in the correct order so that the resulting copy is correct; the data in the original location is changed only when the two areas overlap.

The *range* parameter specifies the starting and ending addresses or the starting address and length of the memory to be copied. The *address* parameter is the first byte in which the copy will be placed. If *range* does not contain an explicit segment, SYMDEB uses DS; if *address* does not contain a segment, SYMDEB uses the same segment used for *range*.

Example

To copy the data in locations DS:0800H through DS:08FFH to locations DS:0900H through DS:09FFH, type

```
-M 800 8FF 900  <Enter>
```

or

```
-M 800 L100 900  <Enter>
```

SYMDEB: N

Name File or Command-Tail Parameters

Purpose

Inserts parameters into the simulated program segment prefix (PSP).

Syntax

N *parameter*[*parameter*...]

where:

parameter is a filename or switch to be placed into the simulated PSP.

Description

The Name File or Command-Tail Parameters (N) command is used to enter one or more parameters into the simulated PSP that is built at the base of the buffer holding the program to be debugged. The N command can also be used before the Load File or Sectors (L) and Write File or Sectors (W) commands to name a file to be read from a disk or written to a disk.

The count of the characters following the N command is placed at DS:0080H in the simulated PSP and the characters themselves are copied into the PSP starting at DS:0081H. The string is terminated by a carriage return (0DH), which is not included in the count. If the second and third parameters follow the naming conventions for MS-DOS files, they are parsed into the default file control blocks (FCBs) in the simulated PSP, at offset 5CH and offset 6CH, respectively. Note that this is different from the N command in DEBUG, which loads the first and second parameters into the default FCBs. (Switches and other filenames specified as parameters are stored in the PSP starting at offset 81H along with the rest of the command line but are not parsed into the default FCBs.)

If the N command line contains only one filename, any parameters placed in the default FCBs by a previous N command are destroyed. If the drive included with the second filename parameter is invalid, the AL register is set to 0FFH. If the drive included with the third filename parameter is invalid, the AH register is set to 0FFH. The existence of a file specified with the N command is not verified until it is loaded with the L command.

The filename at DS:0081H specifies the file that is read or written by a subsequent L or W command.

Example

Assume that SYMDEB was started without specifying the name of a target program in the command line. To load the program CLEAN.COM for execution under the control of

SYMDEB and include the parameter MYFILE.DAT in the simulated PSP's command tail and FCB, use the N and L commands together as follows:

```
-N CLEAN.COM  MYFILE.DAT  <Enter>
-L  <Enter>
```

To execute the program CLEAN.COM, type

```
-G  <Enter>
```

The net effect is the same as if the CLEAN.COM program had been run from the MS-DOS command level with the command line

```
C>CLEAN MYFILE.DAT  <Enter>
```

except that the program is executing under the control of SYMDEB and within SYMDEB's memory buffer.

SYMDEB: O

Output to Port

Purpose

Writes 1 byte to an input/output (I/O) port.

Syntax

O *port byte*

where:

port	is a 16-bit I/O port address in the range 0 through FFFFH.
byte	is a value to be written to the I/O port (0–0FFH).

Description

The Output to Port (O) command writes 1 byte of data to the specified I/O port address. The data value must be in the range 00H through 0FFH.

Warning: This command must be used with caution because it involves direct access to the computer hardware and no error checking is performed. Attempts to write to some port addresses, such as those for ports connected to peripheral device controllers, timers, or the system's interrupt controller, may cause the system to crash or may even result in damage to data stored on disk.

Example

To write the value C8H to I/O port 10AH, type

```
-O 10A C8  <Enter>
```

SYMDEB: P

Proceed Through Loop or Subroutine

Purpose

Executes a loop, string instruction, software interrupt, or subroutine to completion.

Syntax

P[=*address*] [*number*]

where:

address is the location of the first instruction to be executed.
number is the number of instructions to execute.

Description

The Proceed Through Loop or Subroutine (P) command transfers control to the target program. The program executes without interruption until the loop, repeated string instruction, software interrupt, or subroutine call at *address* is completed or until the specified number of machine instructions have been executed. Control then returns to SYMDEB and the current contents of the target program's registers and flags are displayed.

Warning: The P command should not be used to execute any instruction that changes the contents of the Intel 8259 interrupt mask (ports 20H and 21H on the IBM PC and compatibles) and cannot be used to trace through ROM. Use the Go (G) command instead.

If the *address* parameter does not contain a segment, SYMDEB uses the target program's CS register; if *address* is omitted, execution begins at the current address specified by the target's CS:IP registers. The *address* parameter must be preceded by an equal sign (=) to distinguish it from *number*.

The *number* parameter specifies the number of instructions to be executed before control returns to SYMDEB. If *number* is omitted, one instruction is executed.

When the Enable Source Display Mode (S+) command is selected, the P command operates directly on source-code lines, passing over function or procedure calls. (The S+ command can be used only with programs created by high-level-language compilers that insert line-number information into object modules.)

When source display mode is disabled with the S– command or when the program being debugged does not have a .SYM file or has been created with the Microsoft Macro Assembler (MASM) or with a compiler that does not support line numbers in relocatable object modules, the P command behaves like the Trace Program Execution (T) command except that when P encounters a loop, repeated string instruction, software interrupt, or subroutine call, it executes it to completion and then returns to the instruction following the

call. For example, if the user wants to trace the first three instructions in a program and if the second instruction is a subroutine call, a P3 command executes the first instruction, goes to the second instruction, identifies it as a CALL instruction, jumps to the subroutine and executes the entire subroutine, comes back and executes the third instruction, and then stops. A T3 command, on the other hand, executes the first instruction, executes the second, executes the first instruction of the subroutine as its third instruction, and then stops. If the instruction at *address* is not a loop, repeated string instruction, software interrupt, or subroutine call, the P command functions just like the T command. After each instruction is executed, SYMDEB displays the current contents of the target program's registers and flags and the next instruction to be executed.

Examples

Assume that the program being debugged was compiled with Microsoft C, a .SYM file was loaded with the executable program to provide line-number information, and source-code display has been enabled with the S+ command. To execute the machine instructions corresponding to the next four lines of source code, type

```
-P 4  <Enter>
```

Assume that the target program was created with MASM and location CS:143FH contains a CALL instruction. To execute the subroutine that is the destination of CALL at full speed and then return control to SYMDEB, type

```
-P =143F  <Enter>
```

SYMDEB: Q

Quit

Purpose

Ends a SYMDEB session.

Syntax

Q

Description

The Quit (Q) command terminates the SYMDEB program and returns control to MS-DOS or the command shell that invoked SYMDEB. Any changes made to a program or other file that were not previously saved to disk with the Write File or Sectors (W) command are lost when the Q command is used.

Example

To exit SYMDEB, type

```
-Q  <Enter>
```

SYMDEB: R

Display or Modify Registers

Purpose

Displays one or all registers and allows a register to be modified.

Syntax

R

or

R *register*[[=] *value*]

where:

register is the two-character name of an Intel 8086/8088 register from the following list:

```
AX BX CX DX SP BP SI DI
DS ES SS CS IP PC
```

or the character F, to indicate the CPU flags.

= is an optional equal sign preceding *value*.

value is a 16-bit integer (0–FFFFH) that will be assigned to the specified register.

Description

The Display or Modify Registers (R) command allows the target program's register contents and CPU flags to be displayed and modified.

If R is entered without a *register* parameter, the current contents of all registers and CPU flags are displayed, followed by a disassembly of the machine instruction currently pointed to by the target program's CS:IP registers.

A register can be assigned a new value in a single command by entering both *register* and *value* parameters, optionally separated by an equal sign (=). If a register is named but no value is supplied, SYMDEB displays the current contents of the specified register and then prompts with a colon character (:) for a new value to be placed in the register. The user can enter the value in any valid radix or as an expression and then press the Enter key. If no radix is appended to the new value, hexadecimal is assumed. If the user presses the Enter key alone in response to the prompt, no changes are made to the register contents.

Note: The PC register name is not supported properly in some versions of SYMDEB, so the IP register name should always be used instead.

Flag Name	Value If Set (1)	Value If Clear (0)
Overflow	OV (Overflow)	NV (No Overflow)
Direction	DN (Down)	UP (Up)
Interrupt	EI (Enabled)	DI (Disabled)
Sign	NG (Minus)	PL (Plus)
Zero	ZR (Zero)	NZ (Not Zero)
Aux Carry	AC (Aux Carry)	NA (No Aux Carry)
Parity	PE (Even)	PO (Odd)
Carry	CY (Carry)	NC (No Carry)

After displaying the current flag values, SYMDEB again displays its prompt (-). Any or all of the individual flags can then be altered by typing one or more two-character flag codes (in any order and optionally separated by spaces) from the list above and then pressing the Enter key. If the user responds to the prompt by pressing the Enter key without entering any codes, no changes are made to the status of the flags.

Examples

To display the current contents of the target program's CPU registers and flags, followed by the disassembled mnemonic for the next instruction to be executed (pointed to by CS:IP), type

```
-R  <Enter>
```

This produces the following display:

```
AX=0000  BX=0000  CX=00A1  DX=0000  SP=FFFE  BP=0000 SI=0000  DI=0000
DS=19A5  ES=19A5  SS=19A5  CS=19A5  IP=0100   NV UP EI PL NZ NA PO NC
19A5:0100 BF8000        MOV   DI,0080
```

If the source display mode is enabled, the R command displays the following:

```
AX=0000  BX=1044  CX=0000  DX=0102  SP=103C  BP=0000  SI=00EA  DI=115E
DS=2143  ES=2143  SS=2143  CS=1F6E  IP=0010   NV UP EI PL ZR NA PE NC
32:    int   argc;
_TEXT:_main:
1F6E:0010 55              PUSH    BP                              ;BR0
```

This format includes the source code that corresponds to the next instruction to be executed.

To set the contents of register AX to FFFFH without displaying its current value, type

```
-R AX=FFFF  <Enter>
```

or

```
-R AX -1  <Enter>
```

To display the current value of the target program's BX register, type

```
-R BX  <Enter>
```

If BX contains 200H, for example, SYMDEB displays that value and then issues a prompt in the form of a colon:

```
BX 0200
:
```

The contents of BX can then be altered by typing a new value and pressing the Enter key, or the contents can be left unchanged by pressing the Enter key alone.

To set the direction and carry flags, first type

```
-R F  <Enter>
```

SYMDEB displays the current flag values, followed by a prompt in the form of a hyphen character (-). For example:

```
NV UP EI PL NZ NA PO NC  -
```

The direction and carry flags can then be set by entering

```
-DN CY  <Enter>
```

on the same line as the prompt.

Messages

Bad Flag!
An invalid code for a CPU flag was entered.

Bad Register!
An invalid register name was entered.

Double Flag!
Two values for the same CPU flag were entered in the same command.

SYMDEB: S

Search Memory

Purpose

Searches memory for a pattern of one or more bytes.

Syntax

S *range list*

where:

range	is the starting and ending address or the starting address and length in bytes of the area to be searched.
list	is one or more byte values or a string to be searched for.

Description

The Search Memory (S) command searches a designated range of memory for a sequence of byte values or text strings and displays the starting address of each set of matching bytes. The contents of the searched area are not altered.

The *range* parameter specifies the starting and ending address or the starting address and length in bytes of the area to be searched. If a segment is not included in *range*, SYMDEB uses DS. If a segment is specified only for the starting address, SYMDEB uses the same segment for the ending address. If a starting address and length in bytes are specified, the starting address plus the length less 1 cannot exceed FFFFH.

The *list* parameter is one or more hexadecimal byte values and/or strings separated by spaces, commas, or tab characters. Strings must be enclosed in single or double quotation marks, and case is significant within a string.

Examples

To search for the string *Copyright* in the area of memory from DS:0000H through DS:1FFFH, type

```
-S 0 1FFF 'Copyright'  <Enter>
```

or

```
-S 0 L2000 "Copyright"  <Enter>
```

If a match is found, SYMDEB displays the address of each occurrence:

```
20A8:0910
20A8:094F
20A8:097C
```

To search for the byte sequence *3BH 06H* in the area of memory from CS:0100H through CS:12A0H, type

```
-S CS:100 12A0 3B 06  <Enter>
```

or

```
-S CS:100 L11A1 3B 06  <Enter>
```

SYMDEB: S+

Enable Source Display Mode

Purpose

Displays source-code lines, rather than machine instructions.

Syntax

S+

Description

The Enable Source Display Mode (S+) command affects the display format of certain SYMDEB commands: Proceed Through Loop or Subroutine (P), Trace Program Execution (T), and Display or Modify Registers (R). The S+ command causes source code, rather than disassembled machine instructions, to be displayed by those commands.

The S+ command is useful only if the program being debugged was created with a high-level-language compiler capable of placing line-number information into the relocatable object modules processed by the Microsoft Object Linker (LINK). When debugging Microsoft Macro Assembler (MASM) programs or programs generated by language compilers that do not pass line-number information to LINK, the S+ command has no effect.

Example

To enable the display of source-code statements during debugging, type

```
-S+  <Enter>
```

SYMDEB: S–

Disable Source Display Mode

Purpose

Displays disassembled machine instructions, rather than source-code lines.

Syntax

S–

Description

The Disable Source Display Mode (S–) command affects the display format of certain SYMDEB commands: Proceed Through Loop or Subroutine (P), Trace Program Execution (T), and Display or Modify Registers (R). The S– command causes disassembled machine instructions, rather than source code, to be displayed by those commands. By default, SYMDEB displays disassembled machine instructions when debugging Microsoft Macro Assembler (MASM) programs or programs generated by language compilers that do not pass line-number information to the Microsoft Object Linker (LINK).

Example

To disable the display of source-code statements during debugging, type

```
-S-  <Enter>
```

SYMDEB: S&

Enable Source and Machine Code Display Mode

Purpose

Displays both source-code lines and disassembled machine instructions.

Syntax

S&

Description

The Enable Source and Machine Code Display Mode (S&) command affects the display format of certain SYMDEB commands: Proceed Through Loop or Subroutine (P), Trace Program Execution (T), and Display or Modify Registers (R). The S& command causes both the disassembled machine instructions and the corresponding source-code lines to be displayed by those commands.

The S& command is useful only if the program being debugged was created with a high-level-language compiler capable of placing line-number information into the relocatable object modules processed by the Microsoft Object Linker (LINK). When debugging Microsoft Macro Assembler (MASM) programs or programs generated by language compilers that do not pass line-number information to LINK, the S& command has no effect.

Example

To enable the display of both source-code statements and disassembled machine-code statements during debugging, type

```
-S&  <Enter>
```

SYMDEB: T

Trace Program Execution

Purpose

Executes one or more machine instructions in single-step mode.

Syntax

T[=*address*] [*number*]

where:

address	is the location of the first instruction to be executed.
number	is the number of machine instructions to be executed.

Description

The Trace Program Execution (T) command executes one or more machine instructions, starting at the specified address. If source display mode has been enabled with the S+ command, each trace operation executes the machine code corresponding to one source statement and displays the lines from the source code. If source display mode has been disabled with the S– command, each trace operation executes an individual machine instruction and displays the contents of the CPU registers and flags after execution.

Warning: The T command should not be used to execute any instruction that changes the contents of the Intel 8259 interrupt mask (ports 20H and 21H on the IBM PC and compatibles). Use the Go (G) command instead.

The *address* parameter points to the first instruction to be executed. If *address* does not include a segment, SYMDEB uses the target program's CS register; if *address* is omitted entirely, execution is begun at the current address specified by the target program's CS:IP registers. The *address* parameter must be preceded by an equal sign (=) to distinguish it from *number*.

The *number* parameter specifies the hexadecimal number of source-code statements or machine instructions to be executed before the SYMDEB prompt is displayed again (default = 1). If source display mode is enabled, the *number* parameter is required. Execution of a sequence of instructions using the T command can be interrupted at any time by pressing Ctrl-C or Ctrl-Break and can be paused by pressing Ctrl-S (pressing any key resumes the trace).

Examples

To execute one instruction at location CS:1A00H and then return control to SYMDEB, displaying the contents of the CPU registers and flags, type

```
-T =1A00  <Enter>
```

Consecutive instructions can then be executed by entering repeated T commands with no parameters.

If source display mode has been enabled with a previous S+ command, to begin execution at the label *main* and continue through the machine code corresponding to four source-code statements, type

```
-T =_main 4  <Enter>
```

SYMDEB: U

Disassemble (Unassemble) Program

Purpose

Disassembles machine instructions into assembly-language mnemonics.

Syntax

U [*range*]

where:

range specifies the starting and ending addresses or the starting address and the number of instructions of the machine code to be disassembled.

Description

The Disassemble (Unassemble) Program (U) command translates machine instructions into their assembly-language mnemonics.

The *range* parameter specifies the starting and ending addresses or the starting address and number of machine instructions to be disassembled. If *range* does not include an explicit segment, SYMDEB uses CS. Note that the resulting disassembly will be incorrect if the starting address does not fall on an 8086 instruction boundary.

If *range* does not include the number of machine instructions to be executed or an ending address, eight instructions are disassembled. If *range* is omitted completely, eight instructions are disassembled starting at the address following the last instruction disassembled by the previous U command, if a U command has been used; if no U command has been used, eight instructions are disassembled starting at the address specified by the current value of the target program's CS:IP registers.

The display format for the U command depends on the current source display mode setting and on whether the program was developed with a compatible high-level-language compiler. If the source display mode setting is S– or the program was developed with the Microsoft Macro Assembler (MASM) or a noncompatible high-level-language compiler, the display contains only the address and the disassembled equivalent of each instruction within *range*. (For 8-bit immediate operands, SYMDEB also displays the ASCII equivalent as a comment following a semicolon.) If the setting is S+ or S& and a compatible symbol file containing line-number information was loaded with the program being debugged, the display contains both the source-code lines and their corresponding disassembled machine instructions.

Note: The 80286 instructions that are considered privileged when the microprocessor is running in protected mode are not supported by SYMDEB's disassembler.

Examples

To disassemble four machine instructions starting at CS:0100H, type

```
-U 100 L4  <Enter>
```

This produces the following display:

```
44DC:0100 EC              IN      AL,DX
44DC:0101 B80200          MOV     AX,0002
44DC:0104 E86102          CALL    0368
44DC:0107 57              PUSH    DI
```

Successive eight-instruction fragments of machine code can be disassembled by entering additional U commands without parameters.

When a program is being debugged with a symbol file that contains line-number information and source display mode has been enabled, disassembled machine code is accompanied by the corresponding source code:

```
43:         if (argc != 2)
28A5:0031 837E0402        CMP     Word Ptr [BP+04],+02
28A5:0035 7503            JNZ     _main+2A (003A)
28A5:0037 E91400          JMP     _main+3E (004E)
44:             {  fprintf(stderr,"\ndump: wrong number of parameters\n");
28A5:003A B83600          MOV     AX,0036
28A5:003D 50              PUSH    AX
28A5:003E B8F600          MOV     AX,00F6
28A5:0041 50              PUSH    AX
28A5:0042 E8AC04          CALL    _fprintf
28A5:0045 83C404          ADD     SP,+04
45:                    return(1);
28A5:0048 B80100          MOV     AX,0001
28A5:004B E9AA00          JMP     _main+E8 (00F8)
```

SYMDEB: V

View Source Code

Purpose

Displays lines from the source-code file for the program being debugged.

Syntax

V *address* [*length*]

or

V [*.sourcefile:linenumber*]

where:

address	is the location of an executable instruction in the target program.
length	is an ending address or the number of source-code lines.
.sourcefile	is the base name of the source file of the program being debugged, preceded by a period (.).
linenumber	is the first literal line number of *.sourcefile* to be displayed.

Description

The View Source Code (V) command displays lines of source code for the program being debugged, beginning at the location specified by *address*. If *address* does not include a segment, SYMDEB uses the target program's CS register.

The optional *length* parameter can be an ending address or an L followed by a hexadecimal number of source-code lines. If *length* is not specified, eight lines of source code are displayed.

If the *.sourcefile* parameter is specified, followed by a colon character (:) and a line number, eight lines of source code are displayed, starting at *linenumber*. If the V command is entered without parameters after the *.sourcefile:linenumber* parameter has been specified, eight lines are displayed from the current source file, beginning with the line after the last line displayed with the V command. The *.sourcefile* parameter must be the name of a high-level-language source file in the current directory. Pathnames and extensions are not supported. The *length* option cannot be used with the *.sourcefile* parameter.

Warning: Specifying a file that does not exist in the current directory may cause the system to crash.

The V command can be used only with programs created by a high-level-language compiler that is capable of placing line-number information into the relocatable object modules processed by the Microsoft Object Linker (LINK). The current source display mode setting (S–, S+, or S&) has no effect on the V command.

Examples

Assume that the program DUMP.EXE is being debugged with the aid of the symbol file DUMP.SYM and that the source file DUMP.C is available in the current directory. To display eight lines of source code beginning at the label *_main*, type

```
-V _main  <Enter>
```

This produces the following output:

```
32:         int   argc;
33:         char  *argv[];
34:
35:     {   FILE *dfile;                        /* control block for input file */
36:         int status = 0;                     /* status returned from file read */
37:         int file_rec = 0;                   /* file record number being dumped */
38:         long file_ptr = 0L;                 /* file byte offset for current rec */
39:         char file_buf[REC_SIZE];            /* data block from file */
```

To view eight lines of source code from the file DUMP.C, beginning with line 20, type

```
-V .DUMP:20  <Enter>
```

Message

Source file for *filename* (cr for none)?
The current directory does not contain the source file specified with the *.sourcefile* parameter. Enter the correct filename or press Enter to indicate no source file.

SYMDEB: W

Write File or Sectors

Purpose

Writes a file or individual sectors to disk.

Syntax

W [*address*]

or

W *address drive start number*

where:

address	is the first location in memory of the data to be written.
drive	is the number of the destination disk drive (0 = drive A, 1 = drive B, 2 = drive C, 3 = drive D).
start	is the number of the first logical sector to be written (0–FFFFH).
number	is the number of consecutive sectors to be written (0–FFFFH).

Description

The Write File or Sectors (W) command transfers a file or individual sectors from memory to disk.

When the W command is entered without parameters or with an address alone, the number of bytes specified by the contents of registers BX:CX are written from memory to the file named by the most recent Name File or Command-Tail Parameters (N) command or to the first file specified in the SYMDEB command line if the N command has not been used.

Note: If a Go (G), Proceed Through Loop or Subroutine (P), or Trace Program Execution (T) command was previously used or the contents of the BX or CX registers were changed, BX:CX must be restored before the W command is used.

When *address* is not included in the command line, SYMDEB uses the target program's CS:0100H. Files with a .EXE or .HEX extension cannot be written with the W command.

The W command can also be used to bypass the MS-DOS file system and obtain direct access to logical sectors on the disk. To use the W command in this way, the memory address (*address*), disk unit number (*drive*), starting logical sector number (*start*), and number of sectors to be written (*number*) must all be provided in the command line in hexadecimal format.

Warning: Extreme caution should be used with the W command. The disk's file structure can easily be damaged if the command is entered incorrectly. The W command should not be used to write logical sectors to network drives.

Example

Assume that the interactive Assemble Machine Instructions (A) command was used to create a program in SYMDEB's memory buffer that is 32 (20H) bytes long, beginning at offset 100H. This program can be written into the file QUICK.COM by sequential use of the Name File or Command-Tail Parameters (N), Display or Modify Registers (R), and Write File or Sectors (W) commands. First, use the N command to specify the name of the file to be written:

```
-N QUICK.COM  <Enter>
```

Next, use the R command to set registers BX and CX to the length to be written. Register BX contains the upper half or most significant part of the length; register CX contains the lower half or least significant part. Type

```
-R CX  <Enter>
```

SYMDEB displays the current contents of register CX and issues a colon character (:) prompt . Enter the length after the prompt:

```
:20  <Enter>
```

To use the R command again to set the BX register to zero, type

```
-R BX  <Enter>
```

Then type

```
:0  <Enter>
```

To create the disk file QUICK.COM and write the program into it, type

```
-W  <Enter>
```

SYMDEB responds:

```
Writing 0020 bytes
```

Messages

EXE and HEX files cannot be written
Files with a .EXE or .HEX extension cannot be written to disk with the W command.

Writing *nnnn* bytes
After a successful write operation, SYMDEB displays in hexadecimal format the number of bytes written to disk.

SYMDEB: X

Examine Symbol Map

Purpose

Displays names and addresses in the symbol maps.

Syntax

X[*]

or

X? [*map*!] [*segment*:] [*symbol*]

where:

map!	is the name of a symbol file, without the .SYM extension, followed by an exclamation point (!).
segment:	is the name of a segment within the currently open or specified *map*, followed by a colon character (:).
symbol	is a symbol name within the specified *segment*.

Description

The Examine Symbol Map (X) command displays the addresses and names of symbols in the currently open symbol maps. (SYMDEB maintains a symbol map for each symbol file specified in the SYMDEB command line.)

If the X command is followed by the asterisk wildcard character (*), the map names, segment names, and segment addresses for all currently loaded symbol maps are displayed. If X is entered alone, the information is displayed only for the active symbol map.

Information from the symbol maps can be displayed selectively by following the X? command with the *map*!, *segment*:, and *symbol* parameters. The three parameters may be used individually or in combination, but at least one parameter must be specified.

The *map*! parameter must be terminated by an exclamation point and consists of the name, without the extension, of a previously loaded symbol file. If *map*! is omitted, SYMDEB uses the currently open symbol map. If more than one .SYM file is specified in the command line, the one with the same name as the program being debugged is opened first.

The *segment*: parameter must be terminated with a colon; it is the name of a segment declared within the specified or currently open symbol map.

The *symbol* parameter is the name of a label, variable, or other object within the specified *segment*.

Any or all parameters can consist of or include the asterisk wildcard character. For example, X?* displays everything in the current map.

Examples

Assume that the program DUMP.EXE is being debugged with the symbol file DUMP.SYM. If the following is typed

```
-X  <Enter>
```

SYMDEB displays:

```
[456E DUMP]
      [456E _TEXT]
       4743 DGROUP
```

This indicates that the program contains one executable code segment (named _TEXT), which is loaded at segment 456EH, and one NEAR DATA group and segment (named DGROUP), which is loaded at segment 4743H.

To display the addresses of all procedures in the same example program whose names begin with the character *f*, type

```
-X? _TEXT:_F*  <Enter>
```

This produces the following listing:

```
_TEXT: (456E)
0428 _fclose          04CB _fopen           04F1 _fprintf
0528 _fread           0ACB _fflush          0BC2 _free
19AD _flushall
```

Note: Unlike the Microsoft C Compiler, SYMDEB is not case sensitive.

SYMDEB: XO

Open Symbol Map

Purpose

Selects the active symbol map and/or segment.

Syntax

XO [*map*!] [*segment*]

where:

map! is the name of a symbol file, without the .SYM extension, followed by an exclamation point (!).

segment is the name of the segment that will become the active segment in the current symbol map.

Description

The Open Symbol Map (XO) command selects the active symbol map and/or the active segment within the current symbol map to be used during debugging.

The optional *map*! parameter must be terminated by an exclamation point and must be the name, without the extension, of a symbol file specified in the original SYMDEB command line. If *map*! is omitted, no changes are made to the active symbol map.

The optional *segment* parameter must be the name of a segment within the current or specified symbol map. All segments in the active symbol map are accessible; the active segment is searched first for symbols specified in other SYMDEB commands. If *segment* is omitted and a new active symbol map is specified, the segment with the smallest address in the new active symbol map will become the active segment.

Examples

Assume that the program SHELL.EXE has been loaded with the two symbol files SHELL.SYM and VIDEO.SYM. To use the information loaded from VIDEO.SYM as the active symbol map for debugging, type

```
-XO VIDEO!  <Enter>
```

Subsequent entry of the command

```
-XO _TEXT  <Enter>
```

causes the segment _TEXT within the symbol map VIDEO to be searched first for symbol names.

Message

Symbol not found
The specified symbol map or segment does not exist.

SYMDEB: Z

Set Symbol Value

Purpose

Assigns a value to a symbol.

Syntax

Z [*map*!] *symbol value*

where:

map!	is the name of a symbol file, without the .SYM extension, followed by an exclamation point (!).
symbol	is an existing symbol name in the active symbol map or in the symbol map specified by *map*!.
value	is the new address of *symbol* (0–FFFFH).

Description

The Set Symbol Value (Z) command allows the address associated with a name in one of the loaded symbol maps to be overridden by a new value.

Note that altering the address of a symbol at debugging time will not affect other addresses or values that were derived from the value of the same symbol at compilation or assembly time.

The optional *map*! parameter must be terminated by an exclamation point and must be the name, without the extension, of a symbol file specified in the original SYMDEB command line. If *map*! is omitted, SYMDEB uses the active symbol map.

The *symbol* parameter specifies the name of a label, variable, or other object in *map*! or the active symbol map.

The *value* parameter specifies a new address to be associated with *symbol*.

To debug programs created with older versions of FORTRAN and Pascal (Microsoft versions earlier than 3.3 or IBM versions earlier than 2.0), the user must start SYMDEB, locate the first procedure of the program being debugged, and then use the Z command to set the address of DGROUP to the current value of the DS register. (Later versions of FORTRAN and Pascal do this by default.)

Examples

To change the segment address for the symbol DGROUP to 5000H, type

```
-Z DGROUP 5000  <Enter>
```

The actual data associated with the label DGROUP must be moved to the new address before debugging can continue.

To change the segment address for the symbol CODE in the inactive symbol map COUNT to 0F00H, type

```
-Z COUNT! CODE F00  <Enter>
```

SYMDEB: <

Redirect SYMDEB Input

Purpose

Redirects input to SYMDEB.

Syntax

< *device*

where:

device is the name of any MS-DOS device or file.

Description

The Redirect SYMDEB Input (<) command causes SYMDEB to read its commands from the specified text file or character device, rather than from the keyboard (CON).

The *device* parameter specifies the name of any MS-DOS device or file from which commands will be read. If the *device* parameter is a filename, the file must be an ASCII text file and each command in the file must be on a separate line.

If input will be taken from a terminal attached to one of the serial communications ports (AUX, COM1, or COM2), the port must be properly configured with the MODE command before the SYMDEB session is started.

When SYMDEB commands are redirected from a file, the last entry in the file must be either the < CON command, which restores the keyboard as the input device, or the Quit (Q) command. Otherwise, SYMDEB will lock and the system will have to be restarted.

Examples

Assume that the text file FILL.TXT contains the following SYMDEB commands:

```
F CS:0100 L100 00
D CS:0100 L100
R
Q
```

To process FILL.TXT during a SYMDEB session (which in turn exits SYMDEB with the Quit [Q] command), type

```
-< FILL.TXT <Enter>
```

Assume that the text file SEARCH.TXT contains the following SYMDEB commands:

```
S BUFFER L2000 "error"
< CON
```

To process SEARCH.TXT during a SYMDEB session and return control to the console, type

```
-< SEARCH.TXT  <Enter>
```

SYMDEB: >

Redirect SYMDEB Output

Purpose

Redirects SYMDEB's output to a device or file.

Syntax

> *device*

where:

device is the name of any MS-DOS device or file.

Description

The Redirect SYMDEB Output (>) command causes SYMDEB to send all its messages to the specified device or file, rather than to the video display (CON). This is useful for creating a record of a debugging session that can be viewed later with an editor or listed on a printer.

After SYMDEB output is redirected, commands typed on the keyboard are not echoed to the video display. Therefore, the user must know in advance which commands to use and which parameters to supply.

The *device* parameter specifies the name of an MS-DOS device or file to receive SYMDEB's output. If output will be redirected to one of the serial communications ports (AUX, COM1, or COM2), the port must be properly configured with the MODE command before the SYMDEB session is started.

Output can be restored to the video display by entering the > CON command or by terminating SYMDEB with the Quit (Q) command.

Examples

To cause SYMDEB to send all prompts and messages to the file SESSION.TXT, type

```
-> SESSION.TXT  <Enter>
```

After this command, new commands are still accepted by SYMDEB, but the keypresses are not echoed to the screen until the command

```
-> CON  <Enter>
```

is entered or SYMDEB is terminated with the Quit (Q) command.

To cause SYMDEB to send all its prompts and messages to the standard printing device, PRN, type

```
-> PRN  <Enter>
```

SYMDEB: =

Redirect SYMDEB Input and Output

Purpose

Redirects both input and output for SYMDEB.

Syntax

= *device*

where:

device is the name of any MS-DOS device.

Description

The Redirect SYMDEB Input and Output (=) command causes SYMDEB to read its commands from and send its output to the specified device, rather than reading from the keyboard and sending output to the video display (CON). This command is especially useful for debugging programs that run in graphics mode; the SYMDEB commands can be entered on a terminal attached to the computer's serial port while the graphics program has the full use of the system's video display.

The *device* parameter specifies the name of any MS-DOS device. If input and output will be redirected to one of the serial communications ports (AUX, COM1, or COM2), the port must be properly configured with the MODE command before the SYMDEB session is started.

Input and output can be restored to the standard settings with the = CON command.

Example

To redirect SYMDEB's input and output to the first serial communications port (COM1), type

```
-= COM1  <Enter>
```

SYMDEB: {

Redirect Target Program Input

Purpose

Redirects input to the program being debugged.

Syntax

{ *device*

where:

device is the name of any MS-DOS device or file.

Description

The Redirect Target Program Input ({) command causes read operations by the program being debugged to be taken from the specified file or device when the program is executed, rather than from the keyboard (CON).

The *device* parameter specifies the name of an MS-DOS device or file from which the target program will read. If the *device* parameter is a filename, the file must be an ASCII text file and each command in the file must be on a separate line.

If input will be taken from a terminal attached to one of the serial communications ports (AUX, COM1, or COM2), the port must be properly configured with the MODE command before the SYMDEB session is started.

Example

To cause input for the program being debugged to be taken from the file TEST.TXT, type

```
-{ TEST.TXT  <Enter>
```

SYMDEB: }

Redirect Target Program Output

Purpose

Redirects the output of the program being debugged.

Syntax

} *device*

where:

device is the name of any MS-DOS device or file.

Description

The Redirect Target Program Output (}) command causes write operations by the program being debugged to be redirected to the specified device or file when the program is executed, rather than to the video display (CON). This is useful for capturing the output of a program in a file for later listing on a printer.

The *device* parameter specifies the name of an MS-DOS device or file to receive the target program's output. If output will be redirected to one of the serial communications ports (AUX, COM1, or COM2), the port must be properly configured with the MODE command before the SYMDEB session is started.

Example

To send the output from the program being debugged to the file SESSION.TXT, type

```
-} SESSION.TXT  <Enter>
```

SYMDEB: ~

Redirect Target Program Input and Output

Purpose

Redirects both input and output for the program being debugged.

Syntax

~ device

where:

device is the name of any MS-DOS device.

Description

The Redirect Target Program Input and Output (~) command causes all read and write operations by the program being debugged to be redirected to the specified character device.

The *device* parameter specifies the name of an MS-DOS device that the target program will read from and write to. If input and output are redirected to one of the serial communications ports (AUX, COM1, or COM2), the port must be properly configured with the MODE command before the SYMDEB session is started.

Example

To redirect input and output for the program being debugged to the first serial communications port (COM1), type

```
-~ COM1  <Enter>
```

SYMDEB: \

Swap Screen

Purpose

Exchanges the SYMDEB display for the target program's display.

Syntax

\

Description

The Swap Screen (\) command causes the SYMDEB status display to be exchanged for the virtual screen used by the program being debugged. After the program's output has been inspected on the virtual screen, the SYMDEB display can be restored by pressing any key. This command is useful for debugging programs that perform direct screen access or run in graphics mode.

Note: Any information on the display when SYMDEB was invoked will also appear on the virtual screen. When SYMDEB is terminated, the current display is set to match the virtual screen.

The Swap Screen command is available only if the /S switch (or the /I switch, if the computer is IBM compatible) preceded the names of the symbol and program files in the original SYMDEB command line.

Example

To exchange the SYMDEB status display for the virtual screen of the program being debugged, type

```
-\  <Enter>
```

To restore the SYMDEB display, press any key.

SYMDEB: .

Display Source Line

Purpose

Displays the current source-code line.

Syntax

.

Description

The Display Source Line (.) command displays the line from the source-code file that corresponds to the machine instruction currently pointed to by the target program's CS:IP registers.

The . command is independent of the current Source Display Mode status (S+, S–, or S&). However, if the program being debugged was not created with a high-level-language compiler that inserts line numbers into the object modules, the . command has no effect.

Example

To display the source-code line corresponding to the next instruction to be executed, type

```
-.  <Enter>
```

This produces output in the following form:

```
56:        printf( '\nDump of file: %s ', argv[1] );
```

SYMDEB: ?

Help or Evaluate Expression

Purpose

Displays the help screen or the value of an expression.

Syntax

? [*expression*]

where:

expression is any valid combination of symbols, addresses, numbers, and operators.

Description

When ? is entered alone, a help screen summarizing all valid SYMDEB commands, operators, and types is displayed.

When ? is followed by the *expression* parameter, *expression* is evaluated and the value is displayed. The *expression* parameter can include any valid combination of symbols, addresses, numbers, and operators.

The form and content of the resulting display depends on the type of expression entered. If *expression* is a symbol or an address (optionally including operators), the value is shown first as a FAR address pointer in the form segment:offset, then as a 32-bit hexadecimal number representing the value's physical location in memory (followed by its decimal equivalent in parentheses), and finally as the physical location's ASCII character equivalents displayed as a string enclosed in quotation marks (which have no practical value if *expression* is an address or symbol).

If *expression* includes numbers (interpreted as signed hexadecimal values unless a radix is specified) and operators, the resulting value is shown first as a 16-bit hexadecimal value, then as a 32-bit hexadecimal value (followed by its decimal equivalent in parentheses), and finally as the value's ASCII character equivalents displayed as a string enclosed in quotation marks.

(The ASCII characters within the string are displayed as dots if their value is less than 20H [32] or greater than 7EH [126].)

Examples

Assume that the pointer array *argv* in the program DUMP.C is located at address 4743:029CH. The command

```
-? _argv+4 <Enter>
```

produces the following display:

```
4743:02A0h  000476D0  (292560)
```

To display the result of an exclusive OR operation between the values 0FCH and 14H, type

```
-? FC XOR 14  <Enter>
```

SYMDEB displays

```
00E8h  000000E8  (232)
```

SYMDEB: !

Escape to Shell

Purpose

Invokes the MS-DOS command processor.

Syntax

! [*command*]

where:

command is the name of any MS-DOS command, program, or batch file and its required parameters.

Description

The Escape to Shell (!) command loads a copy of the system's command processor (COMMAND.COM), optionally passing it the name of a program or batch file to be executed. This allows MS-DOS functions such as listing or copying files to be carried out without losing the context of the debugging session.

If the ! command is entered alone, an additional copy of COMMAND.COM gains control and displays the system prompt. Control can be returned to SYMDEB by leaving the new shell with the EXIT command.

If the ! character is followed by a *command* parameter that specifies any valid MS-DOS command, program name, or batch-file name, the specified command is executed immediately and control returns directly to SYMDEB.

The SYMDEB statement connector (;) cannot be used on the same line as the ! command; all text encountered after this command is passed to COMMAND.COM and is interpreted as an MS-DOS command line.

Example

To list the files in the current directory, type

```
-! DIR /W  <Enter>
```

Messages

COMMAND.COM not found!
SYMDEB could not find COMMAND.COM because it was not present in the directory location specified in the environment block's COMSPEC variable.

Not enough memory!
Free memory in the transient program area (TPA) is insufficient to execute the requested command or program. This is a common occurrence when debugging a large program with symbol files.

SYMDEB: *

Enter Comment

Purpose

Allows insertion of a comment that will be ignored by SYMDEB's command interpreter.

Syntax

**text*

where:

text is any ASCII text up to and including a carriage return.

Description

The Enter Comment (*) command causes the remainder of the text on that line to be ignored, thereby providing a means of commenting a SYMDEB debugging session. SYMDEB echoes any text following the asterisk to the screen or redirected output device, providing the user with a convenient way to comment program output redirected to a file or a printer. A maximum of 78 characters can be included on each comment line. Comment lines are also useful for documenting lines within a text file that SYMDEB will use as redirected input for the program being debugged.

Example

To echo the reminder *Errors in program output start here:* to the screen or redirected output device, type

```
-*Errors in program output start here:  <Enter>
```

A line in a text file that will be used by SYMDEB for redirected input to the program being debugged may be "commented out" by inserting an asterisk at the beginning of the line. For example:

```
*EB CS:1200 90
```

CodeView

Window-Oriented Debugger

Purpose

Allows the controlled execution of an assembly-language program or high-level-language program for debugging purposes. Both source code and the corresponding unassembled machine code can be displayed as program execution is traced. In addition, watch variables, CPU registers and flags, and program output can be examined in separate debugging windows. CodeView is supplied with the Microsoft Macro Assembler (MASM), C Compiler, Pascal Compiler, and FORTRAN Compiler. This documentation describes CodeView version 2.0.

Syntax

CV [*options*] *exe_file* [*parameters*]

where:

exe_file is the name of the executable file containing the program to be debugged (default extension = .EXE).

parameters is one or more filenames or switches required by the program being debugged.

options is one or more switches from the following list. Switches can be either uppercase or lowercase and can be preceded by a dash (–) instead of a forward slash (/).

/2	Allows the use of two video displays for debugging.
/43	Enables 43-line display mode. (An IBM-compatible computer with an enhanced graphics adapter [EGA] and an enhanced color display is required for this option.)
/B	Forces the attached monitor to use two shades of color when displaying information.
/C*commands*	Executes the specified list of startup commands when CodeView is invoked. If the list of startup commands contains any spaces, the entire list must be enclosed in double quotation marks ("). Commands in the list must be separated by a semicolon character (;).
/D	Turns off nonmaskable interrupt trapping and Intel 8259 interrupt trapping. (This switch prevents system crashes on some IBM-compatible machines that do not support certain IBM-specific interrupt trapping functions.)

(more)

/E	Stores the symbolic information of the program in expanded memory.
/F	Enables the screen-flipping method of switching between the debugging display and the virtual output display. Screen flipping is the default method for IBM-compatible computers with color/graphics adapters.
/I	Enables nonmaskable interrupt trapping and Intel 8259 interrupt trapping on computers that are not IBM-compatible.
/M	Disables mouse support within CodeView.
/P	Enables palette register restore mode, which allows non-IBM EGAs to restore the proper colors upon return from the virtual output screen.
/R	Enables Intel 80386 debugging registers.
/S	Enables the screen-swapping method of switching between the debugging display and the virtual output display. Screen swapping is the default method for IBM-compatible computers with monochrome adapters.
/T	Disables window mode. This switch is necessary for some non-IBM computers or when a sequential debugging session is desired.
/W	Enables window mode. This switch allows CodeView to operate in multiple windows on the same screen. (This option is not the default for some computers.)

Description

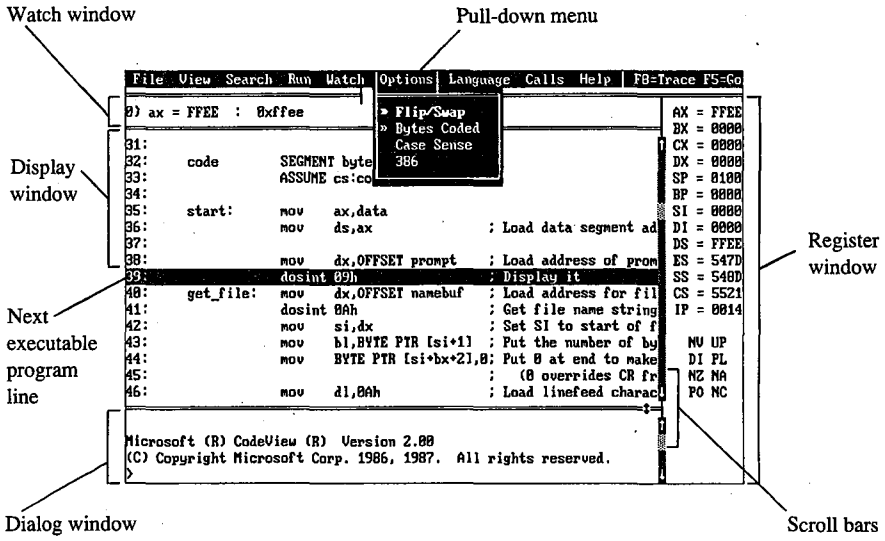
CodeView is a window-oriented menu-driven debugger that allows tracing and debugging of high-level-language programs and assembly-language programs. In general, any valid C, FORTRAN, BASIC, Pascal, or MASM source code can be debugged with CodeView.

To prepare a program for debugging under CodeView, the program must be compiled and linked so that the resulting executable file has the extension .EXE and contains line-number information, a symbol table, and executable code. (To a limited extent, text files and .COM files can also be examined under CodeView.) During the debugging session, the program source file must remain in the current directory if source-code display is desired.

The CodeView screen contains four windows that display information about the program being debugged: the display window, which contains program source code and (if requested) the unassembled machine code corresponding to the source code; the dialog window, where line-oriented commands similar (and in some cases identical) to SYMDEB can be entered and viewed (*see* PROGRAMMING UTILITIES: SYMDEB); the register window (optional), which contains the current status of the microprocessor's registers and flags; and the watch window (optional), which contains program variables or memory

locations to be examined during program execution. CodeView also provides a virtual output screen (stored internally) that contains all display output generated during the CodeView session.

A typical CodeView debugging screen looks like this:



The CodeView display.

Display window commands

Commands that control the display window are available in nine pull-down menus whose names appear in a menu bar near the top of the screen. Commands can be selected with the keyboard or the mouse. Commands are selected with the keyboard by pressing the Alt key, pressing the first letter in the menu name, and then pressing the first letter of the command. Commands are selected with the mouse by pulling down the menu with the mouse pointer, highlighting the command, and then releasing the mouse button. Commands with small double arrows to the left of the command name are currently active. The CodeView menus and commands are described below.

File menu

The File menu includes commands that manipulate the current source or program file. To select the File menu with the keyboard, press Alt-F.

Command	Action
Open...	Opens the specified source file, *include* file, or text file in the display window.
DOS Shell	Exits to the shell temporarily. Type *exit* to return to CodeView.
Exit	Ends the current CodeView session.

View menu

The View menu includes commands that select source or assembly modes and commands that select the debugging screen or the virtual output screen. To select the View menu with the keyboard, press Alt-V.

Command	Action
Source	Displays only the high-level-language or assembly-language source code corresponding to the program being debugged.
Mixed	Displays both the unassembled machine code and the source code corresponding to the program being debugged.
Assembly	Displays only the unassembled machine code corresponding to the program being debugged.
Registers	Displays or removes the optional register window.
Output	Replaces the debugging screen with the virtual output screen. Press any key to return to the debugging screen.

Search menu

The Search menu includes commands that search through text files for text strings and through executable code for labels. To select the Search menu with the keyboard, press Alt-S.

Command	Action
Find...	Searches the current source file or other text file for the specified expression.
Next	Searches forward through the file for the next match of the last expression specified with the Find... command.
Previous	Searches backward through the file for the next match of the last expression specified with the Find... command.
Label...	Searches the executable code for the specified procedure name or program label.

Run menu

The Run menu includes commands that run the program being debugged. To select the Run menu with the keyboard, press Alt-R.

Command	Action
Start	Runs the program at full speed from the first instruction.
Restart	Reloads the program and moves to the first instruction.
Execute	Runs the program at reduced speed from the current instruction.
Clear Breakpoints	Clears all breakpoints.

Watch menu

The Watch menu includes commands that add watch statements to and delete watch statements from the watch window. Watch statements describe expressions or areas of memory to be examined during program execution. To select the Watch menu with the keyboard, press Alt-W.

Command	Action
Add Watch...	Adds the specified watch-expression statement to the watch window.
Watchpoint...	Adds the specified watchpoint statement to the watch window. A watchpoint is a conditional breakpoint that is taken when the expression becomes nonzero (true).
Tracepoint...	Adds the specified tracepoint statement to the watch window. A tracepoint is a conditional breakpoint that is taken when a given expression or range of memory changes.
Delete Watch...	Deletes the specified statement from the watch window.
Delete All Watch	Deletes all statements from the watch window.

Options menu

The Options menu contains commands that affect the general behavior of CodeView. To select the Options menu with the keyboard, press Alt-O.

Command	Action
Flip/Swap	When on (the default), enables screen swapping or screen flipping (whichever option CodeView was started with); when off, disables swapping or flipping. Either method can be used to display the CodeView virtual output screen.
Bytes Coded	When on (the default), displays the instructions, instruction addresses, and the bytes for each instruction; when off, displays only the instructions.
Case Sense	When on, causes CodeView to assume that symbol names are case sensitive; when off, causes CodeView to assume that symbol names are not case sensitive. This option is on by default for C programs and off by default for FORTRAN, BASIC, and assembly programs.
386	When on, allows instructions that reference 32-bit instructions to be assembled and executed and the register window to display 32-bit values. When off, does not allow Intel 80386 instructions and registers to be supported.

Language menu

The Language menu contains commands that select the language-dependent expression evaluator or instruct CodeView to select it for you. To select the Language menu with the keyboard, press Alt-L.

Command	Action
Auto	Forces CodeView to select the expression evaluator of the source file being loaded, based on the extension of the source file.
Basic	Uses a BASIC expression evaluator to determine the value of source-level expressions.
C	Uses a C expression evaluator to determine the value of source-level expressions.
Fortran	Uses a FORTRAN expression evaluator to determine the value of source-level expressions.

Calls menu

The Calls menu is different from other menus in that its contents vary depending on the status of the program. The Calls menu lists the names of specific routines that will be displayed on the screen when that routine name is selected. Routine names in the Calls menu can be selected by typing the number displayed immediately to the left of a routine name. The cursor will move to the line at which the selected routine was last executing.

The current value of each parameter, if any, is shown in parentheses following the name of the routine in the Calls menu. The menu expands to accommodate the parameters of the widest line. Parameters are shown in the current radix (default = decimal). If the program contains more active routines than will fit on the screen or if the routine parameters are too wide, the menu expands to the left and right.

To select the Calls menu with the keyboard, press Alt-C.

Help menu

The Help menu lists the major topics in the CodeView "linked-list" help system. For help, pull down the Help menu and then select the topic of interest. To select the Help menu with the keyboard, press Alt-H.

Command	Action
Intro to Help	Displays information about the "linked-list" help system.
Keyboard/Mouse	Displays information about keyboard and mouse commands.
Run commands	Displays information about Run commands.
Display cmds.	Displays information about Display commands.
Watch/Break	Displays information about setting, listing, and deleting watchpoints and breakpoints.
Memory Ops	Displays information about viewing and modifying memory.
System cmds.	Displays information about system and environment commands.
About CodeView	Displays information about the current CodeView version, time, and date.

Key commands

CodeView supports a variety of function keys and key combinations that modify the active window.

Key	Action
F1	Displays the introductory help screen.
F2	Displays or removes the register window.
F3	Changes the display in the display window to source, mixed, or assembly mode.
F4	Displays the virtual output screen (press any key to return).
F5	Executes to the next breakpoint or to the end of the program if no breakpoint is encountered.
F6	Toggles between the display window and the dialog window.
F7	Sets a temporary breakpoint on the line containing the cursor and executes to that line (or the next breakpoint).
F8	Executes a trace command, stepping through program calls if present.
F9	Sets or clears a breakpoint on the line containing the cursor.
F10	Executes the next source line (in source mode) or the next instruction (in assembly mode), stepping over program calls if present.
Ctrl+G	Increases the size of the display window or the dialog window, whichever is active.
Ctrl+T	Decreases the size of the display window or the dialog window, whichever is active.

Dialog window commands

After CodeView and the specified executable file are loaded, CodeView displays its special prompt character (>) at the bottom of the dialog window and awaits a dialog command. CodeView dialog commands consist of one, two, or three characters, usually followed by one or more parameters. CodeView treats uppercase and lowercase characters the same except when they are contained in strings enclosed within single or double quotation marks. The default radix for dialog command parameters is 10 (decimal). Dialog commands are executed when the Enter key is pressed.

A detailed explanation of CodeView dialog commands and parameters is not presented in this entry. CodeView dialog commands and parameters are similar to SYMDEB commands and parameters. *See* PROGRAMMING UTILITIES: SYMDEB. Additional information about using CodeView dialog commands and parameters can be found in the CodeView documentation supplied with the Microsoft Macro Assembler (MASM), C Compiler, Pascal Compiler, and FORTRAN Compiler. A sample debugging session using CodeView dialog commands and window commands is documented in this book. *See* PROGRAMMING IN THE MS-DOS ENVIRONMENT: PROGRAMMING TOOLS: Debugging in the MS-DOS Environment.

The dialog commands available with CodeView are as follows:

Command	Syntax	Action
!	! [*command*]	Escape to shell.
"	"	Pause redirected file execution.
#	#*number*	Set display window tabs.
*	**comment*	Echo comment to output device.
.	.	Display current source line.
/	/[*searchtext*]	Search for regular expression.
7	7	Display 8087 registers.
:	:[:]...[:]	Delay redirected file execution.
<	< *device*	Redirect dialog window input.
=	= *device*	Redirect dialog window input and output.
>	[T] > [>] device	Redirect dialog window output.
?	? *expression*[,*format*]	Evaluate expression.
@	@	Redraw screen.
A	A [*address*]	Assemble machine instructions.
BC	BC [*] [*list*]	Clear breakpoints.
BD	BD [*] [*list*]	Disable breakpoints.
BE	BE [*] [*list*]	Enable breakpoints.
BL	BL	List breakpoints.
BP	BP [*address* [*passcount*] ["*cmds*"]]	Set breakpoints.
C	C *range address*	Compare memory areas.
D	D [*range*]	Display (dump) memory.
DA	DA [*range*]	Display ASCII.
DB	DB [*range*]	Display bytes.
DD	DD [*range*]	Display doublewords.
DI	DI [*range*]	Display integers.
DL	DL [*range*]	Display long reals.
DS	DS [*range*]	Display short reals.
DT	DT [*range*]	Display 10-byte reals.
DU	DU [*range*]	Display unsigned integers.
DW	DW [*range*]	Display words.
E	E *address* [*list*]	Enter data.
EA	EA *address* [*list*]	Enter ASCII string.
EB	EB *address* [*list*]	Enter bytes.
ED	ED *address* [*value*]	Enter doublewords.
EI	EI *address* [*list*]	Enter integers.
EL	EL *address* [*value*]	Enter long reals.
ES	ES *address* [*value*]	Enter short reals.
ET	ET *address* [*value*]	Enter 10-byte reals.

(more)

Command	Syntax	Action
EU	EU *address* [*value*]	Enter unsigned integers.
EW	EW *address* [*value*]	Enter words.
F	F *range list*	Fill memory.
G	G [*breakpoint*]	Go execute program.
H	H	Display help screen.
I	I *port*	Input from port.
K	K [*number*]	Perform stack trace.
L	L [*parameters*]	Reload program.
M	M *range address*	Move (copy) data.
N	N [*radix*]	Change current radix.
O	O *port byte*	Output to port.
O	O	Display all options.
O3	O3[+|-]	Toggle Intel 80386 option.
OB	OB[+|-]	Toggle bytes coded option.
OC	OC[+|-]	Toggle case-sense option.
OF	OF[+|-]	Toggle flip/swap option.
P	P [*count*]	Step through program (over calls).
Q	Q	Quit debugger.
R	R [*register* [*value*]]	Display or modify registers.
RF	RF [*flags*]	Display or modify flags.
S	S *range list*	Search memory.
S	S	Display current display mode.
S+	S+	Display source code.
S–	S–	Display assembly language.
S&	S&	Display source code and assembly language.
T	T [*count*]	Trace program execution (through calls).
TP	TP [*type*] *range*	Set memory-tracepoint statement.
TP?	TP? *expression*[,*format*]	Set tracepoint-expression statement.
U	U [*range*]	Disassemble (unassemble) program.
USE	USE [*language*]	Switch expression evaluators.
V	V [.[*filename*:]*linenumber*]	View source code.
W	W	List watchpoints and tracepoints.
W	W [*type*] *range*	Set memory-watch statement.
W?	W? *expression*[,*format*]	Set watch-expression statement.
WP?	WP? *expression*[,*format*]	Set watchpoint.
X	X[?[*module*!] [*routine*.]*symbol*|*]	Examine program symbols.
Y	Y [*] [*list*]	Delete watch statements.
\	\	Display virtual output screen.

Examples

To prepare the source file SHELL.C for debugging with CodeView, first compile the source file with the switches that disable optimization and cause symbol-table and line-number information to be written to the relocatable object module:

```
C>MSC /Zi /Od SHELL;  <Enter>
```

Next, to convert the object module to an executable program and prepare it for CodeView, type

```
C>LINK /CO SHELL;  <Enter>
```

To begin debugging, type

```
C>CV SHELL  <Enter>
```

To start CodeView in 43-line mode with TEST.EXE as the executable file and INFO.DAT as the command-tail parameter, type

```
C>CV /43 TEST INFO.DAT  <Enter>
```

In both examples the source file corresponding to the specified executable file must be in the current directory if source-code display is desired.

Messages

Argument to IMAG/DIMAG must be simple type
An invalid parameter to an IMAG or DIMAG function, such as an array with no subscripts, was specified.

Array must have subscript
An array without any subscripts was specified in an expression, such as *IARRAY+2*. A correct example is *IARRAY[1]+2*.

Bad address
An invalid address was specified. For example, an address containing hexadecimal characters might have been specified when the radix is decimal.

Bad breakpoint command
An invalid breakpoint number was specified with the BC, BD, or BE dialog command. The breakpoint number must be in the range 0 through 19.

Bad flag
An invalid flag mnemonic was specified with the RF dialog command.

Bad format string
An invalid format specifier was used following an expression. Expressions used with the ?, W?, WP?, and TP? dialog commands can have format specifiers set off from the expression by a comma. The valid format specifiers are c, d, e, E, f, g, G, i, o, s, u, x, and X. Some format specifiers can be preceded by the prefix h (to specify a 2-byte integer) or l (to specify a 4-byte integer).

Bad integer or real constant
An invalid numeric constant was specified in an expression.

Bad intrinsic function
An invalid intrinsic function name was specified in an expression.

Badly formed type
The type information in the symbol table of the file being debugged is incorrect. This is a serious problem. Note the circumstances of the failure and notify Microsoft Corporation.

Bad radix (use 8, 10, or 16)
An invalid radix was specified with the N dialog command. Use an octal, decimal, or hexadecimal radix.

Bad register
An invalid register name was specified with the R dialog command. Use AX, BX, CX, DX, SP, BP, SI, DI, DS, ES, SS, CS, or IP. If your machine is equipped with an Intel 80386 microprocessor, use EAX, EBX, ECX, EDX, ESP, EBP, ESI, EDI, DS, ES, FS, GS, SS, CS, or IP.

Bad subscript
An invalid subscript expression was specified for an array, such as *IARRAY(3.3)* or *IARRAY((3,3))*. The correct expression for this example (in BASIC or FORTRAN) is *IARRAY (3,3)*.

Bad type cast
Incompatible types of operands were specified in an expression.

Bad type (use one of 'ABDILSTUW')
An invalid type was used in a Display (D, DA, DB, DF, DU, DW, DD, DS, DL, or DT) dialog command. The valid types are ASCII (A), byte (B), integer (I), unsigned (U), word (W), doubleword (D), short real (S), long real (L), and 10-byte real (T).

Breakpoint # or '*' expected
The BC, BD, or BE dialog command was entered without a parameter.

Cannot cast complex constant component into REAL
An incompatible real or imaginary component was specified in a COMPLEX constant. Both real and imaginary components must be compatible with type REAL.

Cannot cast IMAG/DIMAG argument to COMPLEX
An invalid parameter was specified with an IMAG or DIMAG function. IMAG and DIMAG parameters must be simple numeric types.

Cannot use struct or union as scalar
A struct or union variable was used as a scalar value in a C expression. Such variables must be followed by a file specifier or preceded by the address-of (&) operator.

Can't find *filename*
CodeView could not find the executable file specified in the command line.

Character constant too long
A character constant that is too long for the FORTRAN expression evaluator was specified. The limit is 126 bytes.

Character too big for current radix
A radix that is larger than the current CodeView radix was specified in a constant. Use the N dialog command to change the radix.

Constant too big
An unsigned constant number larger than 4,294,967,295 (FFFFFFFFH) was specified.

CPU not an 80386
The 386 option was selected but a machine without an Intel 80386 microprocessor is being used.

Divide by zero
An expression in a parameter of a dialog command attempted to divide by zero.

EMM error
CodeView failed to use the Expanded Memory Manager (EMM) correctly. This is a serious problem. Note the circumstances of the failure and notify Microsoft Corporation.

EMM hardware error
The Expanded Memory Manager (EMM) routines reported a hardware error. Check your expanded memory board for defects.

EMM memory not found
The /E option was used but expanded memory has not been installed. Install software that accesses the memory according to the Lotus/Intel/Microsoft Expanded Memory Specification (LIM EMS).

EMM software error
The Expanded Memory Manager (EMM) routines reported a software error. Reinstall the EMM software.

Expression too complex
An expression given as a dialog-command parameter is too complex.

Extra input ignored
Too many parameters were specified with a command. CodeView evaluates the valid parameters and ignores the rest. In this situation, CodeView often does not evaluate the parameters as intended.

Flip/Swap option off — application output lost
The program being debugged is writing to the screen, but the output cannot be displayed because the flip/swap option has been disabled.

Floating point error
This is a serious problem. Note the circumstances of the failure and notify Microsoft Corporation.

Illegal instruction
This message usually indicates that a machine instruction attempted to divide by zero.

Index out of bound
A subscript value was specified that is outside the bounds declared for the array.

Insufficient EMM memory
Expanded memory is insufficient to hold the program's symbol table.

Internal debugger error
This is a serious problem. Note the circumstances of the failure and notify Microsoft Corporation.

Invalid argument
An invalid CodeView expression was specified as a parameter.

Invalid executable file format — please relink
The executable file was not linked with the version of LINK released with this version of the CodeView debugger. Relink with the appropriate version of LINK.

Invalid option
An invalid switch was specified with the O command.

Missing '"'
A string specified as a parameter to a dialog command did not have a closing double quotation mark.

Missing '('
A parameter to a dialog command was specified as an expression containing a right parenthesis but no left parenthesis.

Missing ')'
A parameter to a dialog command was specified as an expression containing a left parenthesis but no right parenthesis.

Missing ']'
A parameter to a dialog command was specified as an expression containing a left bracket but no right bracket, or a regular expression was specified with a right bracket but no left bracket.

Missing '(' in complex constant
An opening parenthesis of a complex constant in an expression was expected but was not found.

Missing ')' in complex constant
A closing parenthesis of a complex constant in an expression was expected but was not found.

Missing ')' in substring
A closing parenthesis of a substring expression was expected but was not found.

Missing '(' to intrinsic
An opening parenthesis for an intrinsic function was expected but was not found.

Missing ')' to intrinsic
A closing parenthesis for an intrinsic function was expected but was not found.

No closing single quote
A character was specified in an expression used as a dialog-command parameter, but the closing single quotation mark is missing.

No code at this line number
A breakpoint was set on a source line that does not correspond to machine code. (In other words, the source line does not contain an executable statement.) For example, the line might be a data declaration or a comment.

No free EMM memory ha lles
CodeView could not find an available EMM handle. Expanded Memory Manager (EMM) software allocates a fixed number of memory handles (usually 256) to be used for specific tasks.

No match of regular expression
No match was found for the regular expression specified with the Search (S) dialog command or with the Find... command from the Search menu.

No previous regular expression
The Previous command was selected from the Search menu, but CodeView found no previous match for the last regular expression specified.

No source lines at this address
The address specified as a parameter for the V dialog command does not have any source lines. For example, it could be an address in a library routine or an assembly-language module.

No such file/directory
The specified file or directory does not exist.

No symbolic information
The executable file specified is not in the CodeView format. The program cannot be debugged in source mode unless the file is created in the CodeView format. The program can be debugged in assembly mode.

Not an executable file
The file specified to be debugged when CodeView started is not an executable file with a .EXE or .COM extension.

Not a text file
An attempt was made to load a file with the Open... command from the File menu or with the V dialog command, but the file is not a text file. CodeView determines if a file is a text file by checking the first 128 bytes for characters that are not in the ASCII ranges 9 through 13 and 20 through 126.

Not enough space
The ! dialog command or the DOS Shell command from the File menu was chosen, but free memory is insufficient to execute COMMAND.COM. Because memory is released by code in the FORTRAN startup routines, this error always occurs if the ! command is used before executing any code. Use any of the code-execution dialog commands (T, P, or G) to execute the FORTRAN startup code; then try the ! command again. This message also occurs with assembly-language programs that do not specifically release memory.

Object too big
A TP? dialog command was entered with a data object (such as an array) that is larger than 128 bytes.

Operand types incorrect for this operation
An operand in a FORTRAN expression had a type incompatible with the operation applied to it. For example, if P is declared as *CHARACTER P (10)*, then *? P+5* would produce this error, because a character array cannot be an operand of an arithmetic operator.

Operator must have a struct/union type
One of the C member-selection operators (–, >, or .) was used in an expression that does not reference an element of a structure or union.

Operator needs lvalue
An expression was specified that does not evaluate to a memory location in an operation that requires one. (An lvalue is an expression that refers to a memory location.) For example, *buffer (count)* is correct; it represents a symbol in memory. However, *I .EQV. 10* is invalid because it evaluates to TRUE or FALSE instead of to a single memory location.

Overlay not resident
An attempt was made to unassemble machine code from a function that is currently not in memory.

Program terminated normally (*exitcode*)
The program terminated execution normally. The number displayed in parentheses is the exit code returned to MS-DOS by the program.

Radix must be between 2 and 36 inclusive
A radix that is outside the allowable range was specified.

Register variable out of scope
An attempt was made to specify a register variable by using the period (.) operator and a routine name.

Regular expression too complex
The regular expression specified is too complex for CodeView to evaluate.

Regular expression too long
The regular expression specified is too long for CodeView to evaluate.

Restart program to debug
The program being debugged has executed to the end.

Simple variable cannot have argument
A parameter to a simple variable was specified in an expression. For example, given the declaration *INTEGER NUM*, the expression *NUM(1)* is not allowed.

Substring range out of bound
A character expression exceeded the length specified in the CHARACTER statement.

Syntax error
An invalid command line was specified for a dialog command, or an invalid assembly-language instruction was entered with the A dialog command.

Too few array bounds given
The bounds specified in an array subscript do not match the array declaration. For example, given the array declaration *INTEGER IARRAY(3,4)*, the expression *IARRAY(1)* would produce this error message.

Too many array bounds given
The bounds specified in an array subscript do not match the array declaration. For example, given the array declaration *INTEGER IARRAY(3,4)*, the expression *IARRAY (I,3,J)* would produce this error message.

Too many breakpoints
An attempt was made to specify more than 20 breakpoints; CodeView permits only 20.

Too many files
Too few file handles were specified for CodeView to operate correctly. Specify more files in your CONFIG.SYS file.

Type clash in function argument
The type of an actual parameter does not match the corresponding formal parameter, or a subroutine that uses alternate returns was called and the values of the return labels in the actual parameter list are not 0.

Type conversion too complex
An attempt was made to typecast an element of an expression in a type other than the simple types or with more than one level of indirection. An example of a complex type would be typecasting to a struct or union type. An example of two levels of indirection is *char***.

Unable to open file
A file specified in a command parameter or in response to a prompt cannot be opened.

Unknown symbol
An identifier that is not in CodeView's symbol table was specified, or a local variable was used in a parameter when not in the routine where the variable is defined, or a subroutine that uses alternate returns was called and the values of the return labels in the parameter list are not 0.

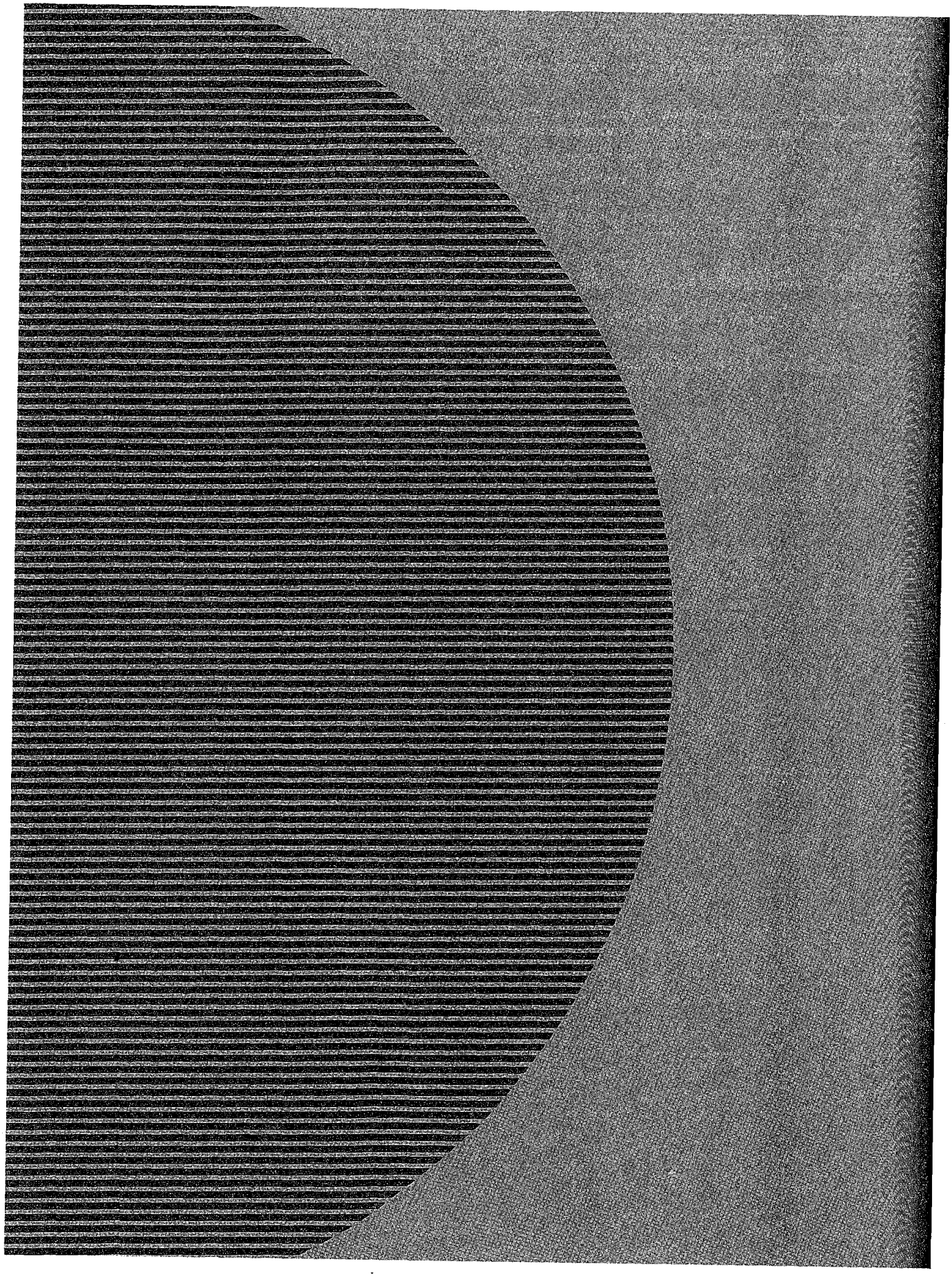
Unrecognized option *option*
Valid options: /B /C<command> /D /E /F /I /M /P /R /S /T /W /43 /2
An invalid switch was entered when starting CodeView.

Usage: cv [options] file [arguments]
An executable file was not specified when starting CodeView.

Video mode changed without /S option
The program changed video modes (either to or from graphics modes) when screen swapping was not specified. Use the /S option to specify screen swapping when debugging graphics programs. Debugging can be continued after receiving this message, but the output screen of the debugged program may be damaged.

Warning: packed file
CodeView was started with a packed file as the executable file. The program cannot be debugged in source mode because all symbolic information is stripped from a file when it is packed with LINK's /EXEPACK option or the EXEPACK utility. Try to debug the program in assembly mode. (The packing routines at the start of the program might make this difficult.)

Wrong number of function arguments
An incorrect number of parameters was specified when evaluating a function in a CodeView expression.



Section V
System Calls

Introduction

All versions of MS-DOS include operating-system services that provide the programmer with hardware-independent tools for handling such tasks as file management, device input and output, memory allocation, and getting and setting system-management information such as the date and time. The majority of these services, collectively called the MS-DOS system calls, are invoked through Interrupt 21H. A few others are called using Interrupts 20H through 27H and 2FH. This section includes descriptions of these system-management services, with details relevant to all releases of MS-DOS through version 3.2.

Use of the Interrupt 21H system calls, rather than hardware-specific routines, helps ensure that a program will run on any computer running an appropriate version of MS-DOS. Likewise, because new releases of MS-DOS attempt to maintain compatibility with earlier versions, use of the calls increases the likelihood that a program will remain usable for more than a single major or minor release of the operating system.

The MS-DOS Interrupt 21H system calls are invoked as follows:

AH	= function number
AL	= subfunction code (if required)
Other registers	= additional function-specific information
Execute Interrupt 21H	

Version Differences

With MS-DOS versions 2.0 and later, considerable overlap occurs in the way in which many system services, such as file and character device I/O, can be carried out. This overlap is a result of the manner in which MS-DOS has developed since it was first released.

The earliest version of MS-DOS, 1.0, included a relatively small set of Interrupt 21H system calls designed primarily for CP/M compatibility. These calls, numbered 00H through 2DH, relied on the use of file control blocks (FCBs) in an application's memory space for information on open files. *See* PROGRAMMING IN THE MS-DOS ENVIRONMENT: PROGRAMMING FOR MS-DOS: File and Record Management; Appendix G: File Control Block (FCB) Structure. The FCB-based system calls in MS-DOS do not support hierarchical file structures, nor do they support redirection of input and output. As a result, many of these system calls have been superseded in later releases of MS-DOS. The CP/M-style calls are no longer recommended and should not be used unless program compatibility with versions 1.x is required.

Beginning with version 2.0, MS-DOS introduced the concept of handles—16-bit numbers returned by the operating system after a successful open or create call. The handles can

subsequently be used by an application program to reference an open file or device, eliminating redundancy and unnecessary overhead. These handles are also used internally by MS-DOS to keep track of open files and devices. The operating system keeps all such handle-related information in its own memory space. Handles offer full support for the hierarchical file system introduced in version 2.0 of MS-DOS and thus allow the programmer to access any file stored in any directory or subdirectory on a block device. Because of the increased flexibility offered by the handle-related system function calls, these services are recommended over the earlier FCB-based calls, which perform similar tasks but for the current directory only. *See* PROGRAMMING IN THE MS-DOS ENVIRONMENT: PROGRAMMING FOR MS-DOS: File and Record Management.

Another advantage of using the system calls introduced in versions 2.0 and later is that these calls set the carry flag when an operational error occurs and return an error code in AX that indicates the nature of the error; the error can then be investigated further by calling Function 59H (Get Extended Error Information). The earlier system calls (00H through 2DH) generally simply return 0FFH (255) in AL to indicate an error or 00H to indicate that the call was completed successfully.

Format of Entries

Entries in this section are arranged in hexadecimal order, with decimal equivalents in parentheses. Each entry is organized as follows:

- Hexadecimal interrupt and/or function number (decimal equivalent in parentheses)
- Interrupt or function name (similar to, but not always the same as, the name used in MS-DOS documentation)
- Version dependencies
- Interrupt or function purpose
- Register contents needed to call
- Register contents on return
- Notes for programmers
- Related functions
- Program example

The format of these entries is designed to give programmers ready reference to specific information, such as register contents, as well as more detailed notes on the use and application of each system call. For further information on the use of the system calls, *see* PROGRAMMING IN THE MS-DOS ENVIRONMENT.

The assembly-language examples in this section use the Cmacros capability introduced with the Windows Software Development Kit. Cmacros, a set of assembly-language macros defined in the file CMACROS.INC, are useful because they provide a simplified interface to the function and segment conventions of high-level languages such as Microsoft C and Microsoft Pascal.

Advantages to using Cmacros for assembly-language programming include transparent support for memory models and symbolic names for function arguments and local variables. Cmacros exist for code and data segment declarations (*sBegin* and *sEnd*), storage allocation (*staticX*, *globalX*, *externX*, and *labelX*), function declarations (*cProc*, *parmX*, *localX*, *cBegin* and *cEnd*), function calls (*cCall*, *Save*, and *Arg*), special definitions (*Def*X, *RegPtr*, and *FarPtr*), and error control (*errnz* and *errn$*). Of these, only *sBegin*, *sEnd*, *cProc*, *parmX*, *localX*, *cBegin*, and *cEnd* are used in the examples in this section.

Two additional macros that support functions not found in CMACROS.INC are *loadCP* and *loadDP*. These macros, included in the file CMACROSX.INC listed below, allow pointers previously declared with *staticX*, *globalX*, *parmX*, *Def*X and *localX* to be loaded into registers without regard to the memory model in use — *loadCP* and *loadDP* generate code to load either the offset portion or the full segment:offset of the address, depending on the memory model.

```
;       CMACROSX.INC
;
;       This file includes supplemental macros for two macros included
;       in CMACROS.INC: parmCP and parmDP. When these macros are used,
;       CMACROS.INC allocates either 1 or 2 words to the variables
;       associated with these macros, depending on the memory model in
;       use. However, parmCP and parmDP provide no support for automatically
;       adjusting for different memory models—additional program code
;       needs to be written to compensate for this. The loadCP and loadDP
;       macros included in this file can be used to provide additional
;       flexibility for overcoming this limit.

;       For example, "parmDP pointer" will make space (1 word in small
;       and middle models and 2 words in compact, large, and huge models)
;       for the data pointer named "pointer". The statement
;       "loadDP ds,bx,pointer" can then be used to dynamically place the
;       value of "pointer" into DS:BX, depending on the memory model.
;       In small-model programs, this macro would generate the instruction
;       "mov dx,pointer" (it is assumed that DS already has the right
;       segment value); in large-model programs, this macro would generate
;       the statements "mov ds,SEG_pointer" and "mov dx,OFF_pointer".


checkDS macro        segmt
           diffcount = 0
           irp d,<ds,DS,Ds,dS>                  ; Allow for all spellings
              ifdif <segmt>,<d>                 ; of "ds".
                 diffcount = diffcount+1
              endif
           endm
           if diffcount EQ 4
              it_is_DS = 0
           else
              it_is_DS = 1
           endif
        endm
```

(more)

```
checkES macro        segmt
          diffcount = 0
          irp d,<es,ES,Es,eS>                    ; Allow for all spellings
             ifdif <segmt>,<d>                   ; of "es".
                diffcount = diffcount+1
             endif
          endm
          if diffcount EQ 4
             it_is_ES = 0
          else
             it_is_ES = 1
          endif
       endm

loadDP  macro        segmt,offst,dptr
          checkDS segmt
          if sizeD                               ; <-- Large data model
             if it_is_DS
                lds  offst,dptr
             else
                checkES segmt
                if it_is_ES
                   les  offst,dptr
                else
                   mov  offst,OFF_&dptr
                   mov  segmt,SEG_&dptr
                endif
             endif
          else
             mov  offst,dptr                     ; <-- Small data model
             if it_is_DS EQ 0
                push ds                          ; If "segmt" is not DS,
                pop  segmt                       ; move ds to segmt.
             endif
          endif
       endm

loadCP  macro        segmt,offst,cptr
          if sizeC                               ; <-- Large code model
             checkDS segmt
             if it_is_DS
                lds offst,cptr
             else
                checkES
                if it_is_ES
                   les  offst,cptr
                else
                   mov  segmt,SEG_&cptr
                   mov  offst,OFF_&cptr
                endif
             endif
          else
```

(more)

```
            push cs                          ; <-- Small code model
            pop  segmt
            mov  offst,cptr
         endif
      endm
```

The following example program demonstrates the use of Cmacros in an assembly-language program:

```
memS    =       0               ;Small memory model
?PLM    =       0               ;C calling conventions
?WIN    =       0               ;Disable Windows support

include cmacros.inc
include cmacrosx.inc

sBegin  CODE                    ;Start of code segment
assumes CS,CODE                 ;Required by MASM

        ;Microsoft C function syntax:
        ;
        ;     int addnums(firstnum, secondnum)
        ;         int firstnum, secondnum;
        ;
        ;Returns firstnum + secondnum

cProc   addnums,PUBLIC          ;Start of addnums functions
parmW   firstnum                ;Declare parameters
parmW   secondnum
cBegin
        mov     ax,firstnum
        add     ax,secondnum
cEnd
sEnd    CODE
        end
```

A simple C program to call this function would be

```
main()
{
        printf("The sum is %d",addnums(12,33));
}
```

Contents by Functional Group

Although distinguishing between FCB-based and handle-based system calls provides a broad and very generalized means of categorizing these services, the more common and useful approach is to group the calls by the type of task they perform. The following list groups the Interrupt 21H system calls and Interrupts 20H, 22H through 27H, and 2FH by type of service.

Function	Purpose
Character Input	
01H	Character Input with Echo
03H	Auxiliary Input
06H	Direct Console I/O
07H	Unfiltered Character Input Without Echo
08H	Character Input Without Echo
0AH	Buffered Keyboard Input
0BH	Check Keyboard Status
0CH	Flush Buffer, Read Keyboard
Character Output	
02H	Character Output
04H	Auxiliary Output
05H	Print Character
06H	Direct Console I/O
09H	Display String
Disk Management	
0DH	Disk Reset
0EH	Select Disk
19H	Get Current Disk
1BH	Get Default Drive Data
1CH	Get Drive Data
2EH	Set/Reset Verify Flag
36H	Get Disk Free Space
54H	Get Verify Flag
File Management	
0FH	Open File with FCB
10H	Close File with FCB
11H	Find First File
12H	Find Next File
13H	Delete File
16H	Create File with FCB
17H	Rename File
1AH	Set DTA Address
23H	Get File Size
2FH	Get DTA Address
3CH	Create File with Handle
3DH	Open File with Handle
3EH	Close File

(more)

Function	Purpose
File Management *(continued)*	
41H	Delete File
43H	Get/Set File Attributes
45H	Duplicate File Handle
46H	Force Duplicate File Handle
4EH	Find First File
4FH	Find Next File
56H	Rename File
57H	Get/Set Date/Time of File
5AH	Create Temporary File
5BH	Create New File
5CH	Lock/Unlock File Region
Information Management	
14H	Sequential Read
15H	Sequential Write
21H	Random Read
22H	Random Write
24H	Set Relative Record
27H	Random Block Read
28H	Random Block Write
3FH	Read File or Device
40H	Write File or Device
42H	Move File Pointer
Interrupt 25H	Absolute Disk Read
Interrupt 26H	Absolute Disk Write
Directory Management	
39H	Create Directory
3AH	Remove Directory
3BH	Change Current Directory
47H	Get Current Directory
Process Management	
00H	Terminate Process
31H	Terminate and Stay Resident
4BH	Load and Execute Program (EXEC)
4CH	Terminate Process with Return Code
4DH	Get Return Code of Child Process
59H	Get Extended Error Information
Interrupt 20H	Terminate Program
Interrupt 27H	Terminate and Stay Resident

(more)

Function	Purpose
Memory Management	
48H	Allocate Memory Block
49H	Free Memory Block
4AH	Resize Memory Block
58H	Get/Set Allocation Strategy
Miscellaneous System Management	
25H	Set Interrupt Vector
26H	Create New Program Segment Prefix
29H	Parse Filename
2AH	Get Date
2BH	Set Date
2CH	Get Time
2DH	Set Time
30H	Get MS-DOS Version Number
33H	Get/Set Control-C Check Flag
34H	Return Address of InDOS Flag
35H	Get Interrupt Vector
38H	Get/Set Current Country
44H	IOCTL
5EH	Network Machine Name/Printer Setup
5FH	Get/Make Assign List Entry
62H	Get Program Segment Prefix Address
63H	Get Lead Byte Table (version 2.25 only)
Interrupt 22H	Terminate Routine Address
Interrupt 23H	Control-C Handler Address
Interrupt 24H	Critical Error Handler Address
Interrupt 2FH	Multiplex Interrupt

Interrupt 20H (32)

1.0 and later

Terminate Program

Interrupt 20H is one of several methods that a program can use to perform a final exit. It informs the operating system that the program is completely finished and that the memory the program occupied can be released.

To Call

CS = segment address of program segment prefix (PSP)

Returns

Nothing

Programmer's Notes

- In response to an Interrupt 20H call, MS-DOS takes the following actions:
  - Restores the termination handler vector (Interrupt 22H) from PSP:000AH.
  - Restores the Control-C vector (Interrupt 23H) from PSP:000EH.
  - With MS-DOS versions 2.0 and later, restores the critical error handler vector (Interrupt 24H) from PSP:0012H.
  - Flushes the file buffers.
  - Transfers to the termination handler address.

  The termination handler releases all memory blocks allocated to the program, including its environment block and any dynamically allocated blocks that were not previously explicitly released; closes any files opened with handles that were not previously closed; and returns control to the parent process (usually COMMAND.COM).
- If the program is returning to COMMAND.COM, control transfers first to COMMAND.COM's resident portion, which reloads COMMAND.COM's transient portion (if necessary) and passes control to it. If a batch file is in progress, the next line of the batch file is then fetched and interpreted; otherwise, a prompt is issued for the next user command.
- Any files that have been written by the program using FCBs should be closed before using Interrupt 20H; otherwise, data may be lost.
- For those programmers who have been with MS-DOS since its earliest incarnations, Interrupt 20H is the traditional way to exit from an application program. However, under versions 2.0 and later, the preferred methods of termination are Interrupt 21H Function 31H (Terminate and Stay Resident) and Interrupt 21H Function 4CH (Terminate Process with Return Code).

Example

```
;************************************************************;
;                                                            ;
;                    Perform a final exit.                   ;
;                                                            ;
;************************************************************;
        int     20H     ; Transfer to MS-DOS.
```

Interrupt 21H (33) Function 00H (0)

1.0 and later

Terminate Process

Function 00H flushes all file buffers to disk, terminates the current process, and releases the memory used by the process.

To Call

AH = 00H
CS = segment of program's program segment prefix (PSP)

Returns

Nothing

Programmer's Notes

- The following interrupt vectors are restored from the PSP of the terminated program:

PSP Offset	Vector for Interrupt
0AH	Interrupt 22H (terminate routine)
0EH	Interrupt 23H (Control-C handler)
12H	Interrupt 24H (critical error handler) (versions 2.0 and later.)

- All file buffers are written to disk and all handles are closed. Control is then transferred to Interrupt 22H (Terminate Routine Address).
- Any file that has changed in length and was opened with an FCB should be closed before Function 00H is called. If such a file is not closed, its length, date, and time are not recorded correctly in the directory.
- With versions 3.x of MS-DOS, restoring the default memory-allocation strategy used by MS-DOS is advisable if that strategy has been changed with Function 58H (Get/Set Allocation Strategy). Any global flags, such as the break and verify flags, that affect system behavior and that have been changed by the process should also be restored to their original values.
- Function 00H performs exactly the same processing as Interrupt 20H (Terminate Program).
- Function 00H is obsolete with MS-DOS versions 2.0 and later. Function 31H (Terminate and Stay Resident) and Function 4CH (Terminate Process with Return Code) are preferred; both enable the terminating process to pass a return code to the calling process and do not require that CS contain the PSP address.

Related Functions

31H (Terminate and Stay Resident)
4CH (Terminate Process with Return Code)

Example

None

Interrupt 21H (33) Function 01H (1)

1.0 and later

Character Input with Echo

Function 01H waits for a character from standard input, echoes it to standard output, and returns the character in the AL register.

To Call

AH = 01H

Returns

AL = 8-bit character code

Programmer's Notes

- With versions 1.x of MS-DOS, Function 01H reads input from the keyboard. With versions 2.0 and later, Function 01H reads a character from standard input, which defaults to the keyboard but can be redirected to another device or to a file. Whether or not input has been redirected, the character is echoed to standard output.
- Function 01H waits for input if a character is not available. A wait can be avoided by calling Function 0BH (Check Keyboard Status), which checks whether a character is available from standard input, and then calling Function 01H if a character is ready.
- On IBM PCs and compatibles, extended characters, such as those produced by the Alt-O and F8 keys, are returned as 2 bytes. The first byte, 00H, signals an extended character; the second byte completes the key code. To read these characters, Function 01H must be called twice.

  With MS-DOS versions 2.0 and later, if standard input has been redirected, the value 00H can also represent a null character from a file and, in that case, might not represent valid data. A program can use Function 44H (IOCTL) Subfunction 00H (Get Device Data) to determine whether standard input has been redirected.
- The carriage-return character (0DH) echoes a carriage return but not a linefeed. Likewise, the linefeed character (0AH) does not echo a carriage return.
- With MS-DOS versions 2.0 and later, Function 01H cannot detect an end-of-file condition if input has been redirected.
- Interrupt 23H (Control-C Handler Address) is called if Control-C (03H) is the input character and (with versions 2.0 and later) input is not redirected.
- With MS-DOS version 2.0 and later, if standard input has been redirected to come from a file, Break must be enabled for Interrupt 23H to be called when Control-C (03H) is the input character.
- Alternative character input functions are 06H (Direct Console I/O), 07H (Unfiltered Character Input Without Echo), and 08H (Character Input Without Echo). The four functions are related as follows:

Function	Waits for Input	Echoes to Std Output	Acts on Control-C
01H	yes	yes	yes
06H	no	no	no
07H	yes	no	no
08H	yes	no	yes

Depending on whether Control-C needs to be filtered, Function 06H, 07H, or 08H can be used to handle character display separately from character input.

- With MS-DOS versions 2.0 and later, Function 3FH (Read File or Device) should be used in preference to Function 01H.

Related Functions

06H (Direct Console I/O)
07H (Unfiltered Character Input Without Echo)
08H (Character Input Without Echo)
0AH (Buffered Keyboard Input)
0CH (Flush Buffer, Read Keyboard)
3FH (Read File or Device)

Example

```
        ;************************************************************;
        ;                                                            ;
        ;           Function 01H: Character Input with Echo          ;
        ;                                                            ;
        ;           int read_kbd_echo()                              ;
        ;                                                            ;
        ;           Returns a character from standard input          ;
        ;           after sending it to standard output.             ;
        ;                                                            ;
        ;************************************************************;

cProc   read_kbd_echo,PUBLIC
cBegin
        mov     ah,01h          ; Set function code.
        int     21h             ; Wait for character.
        mov     ah,0            ; Character is in AL, so clear high
                                ; byte.
cEnd
```

Interrupt 21H (33) Function 02H (2)

1.0 and later

Character Output

Function 02H sends a character to standard output.

To Call

AH = 02H
DL = 8-bit code for character to be output

Returns

Nothing

Programmer's Notes

- With versions 1.x of MS-DOS, Function 02H sends a character to the active display. With MS-DOS versions 2.0 and later, Function 02H sends the character to standard output. By default, the output is sent to the active display, but it can be redirected to another device or to a file.
- With all versions of MS-DOS, displaying a backspace (08H) moves the cursor back one position but does not erase the character at the new position.
- If a Control-C is detected after the character is sent, Interrupt 23H (Control-C Handler Address) is called.
- With MS-DOS versions 2.0 and later, Function 40H (Write File or Device) should be used in preference to Function 02H.

Related Functions

06H (Direct Console I/O)
09H (Display String)
40H (Write File or Device)

Example

```
;*************************************************************;
;                                                             ;
;               Function 02H: Character Output                ;
;                                                             ;
;               int disp_ch(c)                                ;
;                   char c;                                   ;
;                                                             ;
;               Returns 0.                                    ;
;                                                             ;
;*************************************************************;
```

(more)

```
cProc   disp_ch,PUBLIC
parmB   c
cBegin
        mov     dl,c            ; Get character into DL.
        mov     ah,02h          ; Set function code.
        int     21h             ; Send character.
        xor     ax,ax           ; Return 0.
cEnd
```

Interrupt 21H (33) Function 03H (3)

1.0 and later

Auxiliary Input

Function 03H waits for a character from the standard auxiliary device and returns the character in the AL register.

To Call

AH = 03H

Returns

AL = 8-bit character code

Programmer's Notes

- With versions 1.x of MS-DOS, Function 03H reads a character from the first serial port. With versions 2.0 and later, Function 03H reads from the standard auxiliary device (AUX), which defaults to COM1.
- Function 03H waits for input until a character is available from the standard auxiliary device.
- Function 03H is not interrupt driven and does not buffer characters received from the standard auxiliary device. As a result, it may not be fast enough for some telecommunications applications and data may be lost.
- A program cannot perform error detection using Function 03H. On IBM PCs and compatibles, error detection is available through the ROM BIOS Interrupt 14H. Another option is to drive the communications controller directly.
- Function 03H does not ensure that auxiliary input is connected and working, nor does it perform any error checking or set up the auxiliary input device. On IBM PCs and compatibles, the standard auxiliary device, normally COM1, is set to 2400 baud, no parity, 1 stop bit, and 8 databits at startup. These parameters can be changed with the MS-DOS MODE command.
- Some auxiliary input devices do not support 8-bit data transmission. This transmission parameter is a characteristic of the device and the communication parameters to which it is set; it is independent of Function 03H.
- If a Control-C is detected at the console, Interrupt 23H (Control-C Handler Address) is called.
- With MS-DOS versions 2.0 and later, Function 3FH (Read File or Device), which handles strings as well as single characters, should be used in preference to Function 03H.

Related Functions

04H (Auxiliary Output)
3FH (Read File or Device)

Example

```
        ;*********************************************************;
        ;                                                         ;
        ;           Function 03H: Auxiliary Input                 ;
        ;                                                         ;
        ;           int aux_in()                                  ;
        ;                                                         ;
        ;           Returns next character from AUX device.       ;
        ;                                                         ;
        ;*********************************************************;

cProc   aux_in,PUBLIC
cBegin
        mov     ah,03h          ; Set function code.
        int     21h             ; Wait for character from AUX.
        mov     ah,0            ; Character is in AL
                                ; so clear high byte.
cEnd
```

Interrupt 21H (33) Function 04H (4)

1.0 and later

Auxiliary Output

Function 04H sends a character to the standard auxiliary device.

To Call

AH = 04H
DL = 8-bit code for character to be output

Returns

Nothing

Programmer's Notes

- With versions 1.x of MS-DOS, Function 04H sends a character to the first serial port. With versions 2.0 and later, Function 04H sends the character to the standard auxiliary device (AUX), which defaults to COM1.
- Function 04H does not ensure that auxiliary output is connected and working, nor does it perform any error checking or set up the auxiliary output device. On IBM PCs and compatibles, the standard auxiliary device, normally COM1, is set to 2400 baud, no parity, 1 stop bit, and 8 databits at startup. These parameters can be changed with the MS-DOS MODE command.
- Function 04H does not return the status of auxiliary output, nor does it return an error code if the auxiliary output device is not ready for data. If the device is busy, Function 04H waits until it is available.
- Interrupt 23H (Control-C Handler Address) is called if a Control-C is detected at the console.
- With MS-DOS versions 2.0 and later, Function 40H (Write File or Device), which manages strings as well as single characters, should be used in preference to Function 04H.

Related Functions

03H (Auxiliary Input)
40H (Write File or Device)

Example

```
        ;************************************************************;
        ;                                                            ;
        ;                Function 04H: Auxiliary Output              ;
        ;                                                            ;
        ;                int aux_out(c)                              ;
        ;                    char c;                                 ;
        ;                                                            ;
        ;                Returns 0.                                  ;
        ;                                                            ;
        ;************************************************************;

cProc   aux_out,PUBLIC
parmB   c
cBegin
        mov     dl,c            ; Get character into DL.
        mov     ah,04h          ; Set function code.
        int     21h             ; Write character to AUX.
        xor     ax,ax           ; Return 0.
cEnd
```

Interrupt 21H (33) Function 05H (5)

1.0 and later

Print Character

Function 05H sends a character to the standard printer.

To Call

AH = 05H
DL = 8-bit code for character to be output

Returns

Nothing

Programmer's Notes

- With versions 1.x of MS-DOS, Function 05H sends a character to the first parallel port (LPT1). With versions 2.0 and later, Function 05H sends the character to the standard printer (PRN), which defaults to LPT1 unless LPT1 has been reassigned with the MS-DOS MODE command. If redirection is in effect, calls to this function send output to the device currently assigned to LPT1.
- Function 05H does not return the status of the standard printer, nor does it return an error code if the standard printer is not ready for characters. If the printer is busy or off line, Function 05H waits until it is available. MS-DOS does, however, perform error checking during the print operation and send any error messages to the standard error device (normally the display).
- If a Control-C is detected at the console, Interrupt 23H (Control-C Handler Address) is called.
- With MS-DOS versions 2.0 and later, Function 40H (Write File or Device) should be used in preference to Function 05H.

Related Function

40H (Write File or Device)

Example

```
;*************************************************************;
;                                                             ;
;               Function 05H: Print Character                 ;
;                                                             ;
;               int print_ch(c)                               ;
;                   char c;                                   ;
;                                                             ;
;               Returns 0.                                    ;
;                                                             ;
;*************************************************************;
```

(more)

```
cProc   print_ch,PUBLIC
parmB   c
cBegin
        mov     dl,c            ; Get character into DL.
        mov     ah,05h          ; Set function code.
        int     21h             ; Write character to standard printer.
        xor     ax,ax           ; Return 0.
cEnd
```

Interrupt 21H (33) Function 06H (6)

1.0 and later

Direct Console I/O

Function 06H reads a character from standard input or writes a character to standard output.

To Call

AH = 06H

For character input:

DL = FFH

For character output:

DL = 00–FEH (8-bit character code)

Returns

If DL was 0FFH on call and a character was ready:

Zero flag is clear.

AL = 8-bit character code

If DL was 0FFH on call and no character was ready:

Zero flag is set.

Programmer's Notes

- With MS-DOS versions 1.x, Function 06H reads a character from the keyboard or sends a character to the display. With versions 2.0 and later, input and output can be redirected; Function 06H reads from the device currently assigned to standard input or sends to the device currently assigned to standard output.
- Function 06H allows all possible characters and control codes with values between 00H and 0FEH to be read or written with standard input and output and with no filtering by the operating system. The rubout character (0FFH, 255 decimal), however, cannot be output with Function 06H; Function 02H (Character Output) should be used instead.
- On IBM PCs and compatibles, extended characters, such as those produced by the Alt-O and F8 keys, are returned as 2 bytes. The first byte, 00H, signals an extended character; the second byte completes the key code. To read these characters, Function 06H must be called twice.

With MS-DOS versions 2.0 and later, if standard input has been redirected, the value 00H can also represent a null character from a file and, in that case, might not represent valid data. A program can use Function 44H (IOCTL) Subfunction 00H (Get Device Data) to determine whether standard input has been redirected.

- If Function 06H is an input request and a Control-C is read, the character is returned as any other character would be. Interrupt 23H (Control-C Handler Address) is not called.
- With MS-DOS versions 2.0 and later, Function 3FH (Read File or Device) and Function 40H (Write File or Device) should be used in preference to Function 06H.

Related Functions

01H (Character Input with Echo)
02H (Character Output)
07H (Unfiltered Character Input Without Echo)
08H (Character Input Without Echo)
09H (Display String)
0AH (Buffered Keyboard Input)
0CH (Flush Buffer, Read Keyboard)
3FH (Read File or Device)
40H (Write File or Device)

Example

```
        ;************************************************************;
        ;                                                            ;
        ;          Function 06H: Direct Console I/O                  ;
        ;                                                            ;
        ;          int con_io(c)                                     ;
        ;              char c;                                       ;
        ;                                                            ;
        ;          Returns meaningless data if c is not 0FFH,        ;
        ;          otherwise returns next character from             ;
        ;          standard input.                                   ;
        ;                                                            ;
        ;************************************************************;

cProc   con_io,PUBLIC
parmB   c
cBegin
        mov     dl,c            ; Get character into DL.
        mov     ah,06h          ; Set function code.
        int     21h             ; This function does NOT wait in
                                ; input case (c = 0FFH)!
        mov     ah,0            ; Return the contents of AL.
cEnd
```

Interrupt 21H (33) Function 07H (7)

1.0 and later

Unfiltered Character Input Without Echo

Function 07H waits for a character from standard input. It does not echo the character to standard output, and it ignores Control-C characters.

To Call

AH = 07H

Returns

AL = 8-bit character code

Programmer's Notes

- With versions 1.x of MS-DOS, Function 07H reads input from the keyboard. With versions 2.0 and later, Function 07H reads a character from standard input. Standard input defaults to the keyboard but can be redirected to another device or to a file.
- Function 07H waits for input if a character is not available. A wait can be avoided by calling Function 0BH (Check Keyboard Status), which checks whether a character is available from standard input, and then calling Function 07H if a character is ready.
- On IBM PCs and compatibles, extended characters, such as those produced by the Alt-O and F8 keys, are returned as 2 bytes. The first byte, 00H, signals an extended character; the second byte completes the key code. To read these characters, Function 07H must be called twice.

  With MS-DOS versions 2.0 and later, if standard input has been redirected, the value 00H can also represent a null character from a file and, in that case, might not represent valid data. A program can use Function 44H (IOCTL) Subfunction 00H (Get Device Data) to determine whether standard input has been redirected.
- Interrupt 23H (Control-C Handler Address) is not called if a Control-C is read. Function 07H simply passes the character back through the AL register. If Control-C checking is required, Function 08H (Character Input Without Echo) should be used instead.
- With MS-DOS versions 2.0 and later, Function 3FH (Read File or Device) should be used in preference to Function 07H.

Related Functions

01H (Character Input with Echo)
06H (Direct Console I/O)
08H (Character Input Without Echo)
0AH (Buffered Keyboard Input)
0CH (Flush Buffer, Read Keyboard)
3FH (Read File or Device)

Example

```
        ;*********************************************************;
        ;                                                         ;
        ;       Function 07H: Unfiltered Character Input          ;
        ;                     Without Echo                        ;
        ;                                                         ;
        ;       int con_in()                                      ;
        ;                                                         ;
        ;       Returns next character from standard input.       ;
        ;                                                         ;
        ;*********************************************************;

cProc   con_in,PUBLIC
cBegin
        mov     ah,07h          ; Set function code.
        int     21h             ; Wait for character, no echo.
        mov     ah,0            ; Clear high byte.
cEnd
```

Interrupt 21H (33) Function 08H (8)

1.0 and later

Character Input Without Echo

Function 08H waits for a character from standard input. The character is not echoed to standard output.

To Call

AH = 08H

Returns

AL = 8-bit character code

Programmer's Notes

- With versions 1.x of MS-DOS, Function 08H reads input from the keyboard. With versions 2.0 and later, Function 08H reads a character from standard input. Standard input defaults to the keyboard but can be redirected to another device or to a file.
- Function 08H waits for input if a character is not available. A wait can be avoided by calling Function 0BH (Check Keyboard Status), which checks whether a character is available, and then calling Function 08H if a character is ready.
- On IBM PCs and compatibles, extended characters, such as those produced by the Alt-O and F8 keys, are returned as 2 bytes. The first byte, 00H, signals an extended character; the second byte completes the key code. To read these characters, Function 08H must be called twice.

  With MS-DOS versions 2.0 and later, if standard input has been redirected, the value 00H can also represent a null character from a file and, in that case, might not represent valid data. A process can use Function 44H (IOCTL) Subfunction 00H (Get Device Data) to determine whether standard input has been redirected.
- If a Control-C is read and (with versions 2.0 and later) input has not been redirected, Interrupt 23H (Control-C Handler Address) is called. To read the Control-C character as data, Function 07H (Unfiltered Character Input Without Echo) should be used.
- Interrupt 23H (Control-C Handler Address) is called if Control-C is the input character, Break is enabled, and (with versions 2.0 and later) standard input has been redirected to come from a file.
- With MS-DOS versions 2.0 and later, Function 3FH (Read File or Device) should be used in preference to Function 08H.

Related Functions

01H (Character Input with Echo)
06H (Direct Console I/O)
07H (Unfiltered Character Input Without Echo)
0AH (Buffered Keyboard Input)
0CH (Flush Buffer, Read Keyboard)
3FH (Read File or Device)

Example

```
        ;*************************************************************;
        ;                                                             ;
        ;    Function 08H:  Unfiltered Character Input Without Echo   ;
        ;                                                             ;
        ;    int read_kbd()                                           ;
        ;                                                             ;
        ;    Returns next character from standard input.              ;
        ;                                                             ;
        ;*************************************************************;

cProc   read_kbd,PUBLIC
cBegin
        mov     ah,08h          ; Set function code.
        int     21h             ; Wait for character, no echo.
        mov     ah,0            ; Clear high byte.
cEnd
```

Interrupt 21H (33) Function 09H (9)

1.0 and later

Display String

Function 09H sends a string of characters to standard output. The string must end with the dollar-sign character ($). All characters up to, but not including, the $ are displayed.

To Call

AH = 09H
DS:DX = segment:offset of string to display

Returns

Nothing

Programmer's Notes

- With MS-DOS versions 1.x, Function 09H sends the string to the display. With versions 2.0 and later, the string is written to standard output. By default, standard output is sent to the display, but it can be redirected to another device or to a file.
- The string can include any valid ASCII characters, including control codes. Sending a dollar sign with this function, however, is not possible.
- Depending on the device currently serving as standard output, characters other than the normally displayable ASCII characters (20H to 7FH) may or may not be displayed. On IBM PCs and most compatibles, extensions to the displayable ASCII character set (character codes 80H to FFH) appear as foreign or graphics characters.
- Display begins at the current cursor position on standard output. After the string is completely displayed, the cursor position is updated to the location immediately following the string.

  On IBM PCs and compatibles, if the end of a line is reached before the string is completely displayed, a carriage return and linefeed are issued and the next character is displayed in the first position of the following line. If the cursor reaches the bottom right corner of the display before the complete string has been sent, the display is scrolled up one line.
- Control characters are often included in the string to be sent. The following sample fragment of code contains carriage returns and linefeeds:

```
msg     db      'Resident part of TSR.COM installed'
        db      0dh, 0ah
        db      'Copyright (c) 19xx Foo Software, Inc.'
        db      0dh, 0ah, 0ah, 0ah
        db      '$'
```

- If a Control-C is detected, Interrupt 23H (Control-C Handler Address) is called.

- With MS-DOS versions 2.0 and later, Function 40H (Write File or Device) should be used in preference to Function 09H.

Related Functions

02H (Character Output)
06H (Direct Console I/O)
40H (Write File or Device)

Example

```
        ;************************************************************;
        ;                                                            ;
        ;                Function 09H: Display String                ;
        ;                                                            ;
        ;                int disp_str(pstr)                          ;
        ;                    char *pstr;                             ;
        ;                                                            ;
        ;                Returns 0.                                  ;
        ;                                                            ;
        ;************************************************************;

cProc   disp_str,PUBLIC,<ds,di>
parmDP  pstr
cBegin
        loadDP  ds,dx,pstr      ; DS:DX = pointer to string.
        mov     ax,0900h        ; Prepare to write dollar-terminated
                                ; string to standard output, but
                                ; first replace the 0 at the end of
                                ; the string with '$'.
        push    ds              ; Set ES equal to DS.
        pop     es              ; (MS-C does not require ES to be
                                ; saved.)
        mov     di,dx           ; ES:DI points at string.
        mov     cx,0ffffh       ; Allow string to be 64KB long.
        repne   scasb           ; Look for 0 at end of string.
        dec     di              ; Scasb search always goes 1 byte too
                                ; far.
        mov     byte ptr [di],'$' ; Replace 0 with dollar sign.
        int     21h             ; Have MS-DOS print string.
        mov     [di],al         ; Restore 0 terminator.
        xor     ax,ax           ; Return 0.
cEnd
```

Interrupt 21H (33) Function 0AH (10)

1.0 and later

Buffered Keyboard Input

Function 0AH collects characters from standard input and places them in a user-specified memory buffer. Input is accepted until either a carriage return (0DH) is encountered or the buffer is filled to one character less than its capacity. The characters are echoed to standard output.

To Call

AH = 0AH
DS:DX = segment:offset of input buffer

Returns

Nothing

Programmer's Notes

- With MS-DOS versions 1.x, Function 0AH reads a string from the keyboard. With versions 2.0 and later, calls to this function read a string from standard input, which defaults to the keyboard but can be redirected to another device or to a file. The MS-DOS editing keys are active during input with this function.
- The buffer pointed to by DS:DX must have the following format:

Byte	Contents
0	Maximum number of characters to read (1–255); this value must be set by the process before Function 0AH is called.
1	Count of characters read (does not include the carriage return); this value is set by Function 0AH before returning to the process.
2–(n+2)	Actual string of characters read, including the carriage return; n = number of bytes read.

- The first byte of the buffer must contain the maximum number of characters the program will accept, including the carriage return at the end. Because the last byte must be a carriage return, the maximum number of bytes this function will actually read is 254. The carriage return is not included in the character count returned by MS-DOS in the second byte of the buffer.
- If the buffer fills to 1 byte less than its capacity, succeeding characters are ignored and a beep is sounded for each keypress until a carriage return is received.
- If a Control-C is detected and (with versions 2.0 and later) input has not been redirected, Interrupt 23H (Control-C Handler Address) is called.
- With versions 2.0 and later, if standard input has been redirected to come from a file, Break must be enabled for Interrupt 23H (Control-C Handler Address) to be called when Control-C is the input character.

- With MS-DOS versions 2.0 and later, if input is redirected, an end-of-file condition goes undetected by Function 0AH.

Related Functions

01H (Character Input with Echo)
06H (Direct Console I/O)
07H (Unfiltered Character Input Without Echo)
08H (Character Input Without Echo)
0CH (Flush Buffer, Read Keyboard)
3FH (Read File or Device)

Example

```
        ;*********************************************************;
        ;                                                         ;
        ;       Function 0AH: Buffered Keyboard Input             ;
        ;                                                         ;
        ;       int read_str(pbuf,len)                            ;
        ;           char *pbuf;                                   ;
        ;           int len;                                      ;
        ;                                                         ;
        ;       Returns number of bytes read into buffer.         ;
        ;                                                         ;
        ;       Note: pbuf must be at least len+3 bytes long.     ;
        ;                                                         ;
        ;*********************************************************;

cProc   read_str,PUBLIC,<ds,di>
parmDP  pbuf
parmB   len
cBegin
        loadDP  ds,dx,pbuf      ; DS:DX = pointer to buffer.
        mov     al,len          ; AL = len.
        inc     al              ; Add 1 to allow for CR in buf.
        mov     di,dx
        mov     [di],al         ; Store max length into buffer.
        mov     ah,0ah          ; Set function code.
        int     21h             ; Ask MS-DOS to read string.
        mov     al,[di+1]       ; Return number of characters read.
        mov     ah,0
        mov     bx,ax
        mov     [bx+di+2],ah    ; Store 0 at end of buffer.
cEnd
```

Interrupt 21H (33) Function 0BH (11)

1.0 and later

Check Keyboard Status

Function 0BH returns a value in AL that indicates whether a character is available from standard input.

To Call

AH = 0BH

Returns

AL	= 00H	no character available
	FFH	one or more characters available

Programmer's Notes

- With MS-DOS versions 1.x, Function 0BH checks the type-ahead buffer for a character. With versions 2.0 and later, if input has been redirected, Function 0BH checks standard input for a character. If input has not been redirected, the function checks the type-ahead buffer.
- Function 0BH does not indicate how many characters are available; it merely indicates whether at least one character is available.
- If the available character is Control-C, Interrupt 23H (Control-C Handler Address) is called.
- Function 0BH does not remove characters from standard input. Thus, if a character is present, repeated calls return 0FFH in AL until all characters in the buffer are read, either with one of the character-input functions (01H, 06H, 07H, 08H, or 0AH) or with Function 3FH (Read File or Device) using the handle for standard input (0).

Related Functions

06H (Direct Console I/O)
44H Subfunction 06H (IOCTL: Check Input Status)

Example

```
;*********************************************************;
;                                                         ;
;            Function 0BH: Check Keyboard Status          ;
;                                                         ;
;            int key_ready()                              ;
;                                                         ;
;            Returns 1 if key is ready, 0 if not.         ;
;                                                         ;
;*********************************************************;
```

(more)

```
cProc   key_ready,PUBLIC
cBegin
        mov     ah,0bh          ; Set function code.
        int     21h             ; Ask MS-DOS if key is available.
        and     ax,0001h        ; Keep least significant bit only.
cEnd
```

Interrupt 21H (33) Function 0CH (12)

1.0 and later

Flush Buffer, Read Keyboard

Function 0CH clears the standard-input buffer and then performs one of the other keyboard input functions (01H, 06H, 07H, 08H, 0AH).

To Call

AH = 0CH
AL = input function number to execute

If AL is 06H:

DL = FFH

If AL is 0AH:

DS:DX = segment:offset of buffer to receive input

Returns

If AL was 01H, 06H, 07H, or 08H on call:

AL = 8-bit ASCII character from standard input

If AL was 0AH on call:

Nothing

Programmer's Notes

- With versions 1.x of MS-DOS, Function 0CH empties the type-ahead buffer before executing the input function specified in AL. With versions 2.0 and later, if input has been redirected to a file, Function 0CH does nothing before carrying out the input function specified in AL; if input was not redirected, the type-ahead buffer is flushed.
- A function number other than 01H, 06H, 07H, 08H, or 0AH in AL simply flushes the standard-input buffer and returns control to the calling program.
- If AL contains 0AH, DS:DX must point to the buffer in which MS-DOS is to place the string read from the keyboard.
- Because the buffer is flushed before the input function is carried out, any Control-C characters pending in the buffer are discarded. If subsequent input is a Control-C, however, Interrupt 23H (Control-C Handler Address) is called if (in versions 2.0 and later) standard input has not been redirected to come from a file.
- With versions 2.0 and later, if standard input has been redirected to come from a file and, after the buffer is flushed, subsequent input is a Control-C character, Interrupt 23H (Control-C handler address) is called only if Break is enabled.
- This function exists to defeat the type-ahead feature if necessary—for example, to obtain input at a critical prompt the user may not have anticipated.

Related Functions

01H (Character Input with Echo)
06H (Direct Console I/O)
07H (Unfiltered Character Input Without Echo)
08H (Character Input Without Echo)
0AH (Buffered Keyboard Input)
3FH (Read File or Device)

Example

```
        ;************************************************************;
        ;                                                            ;
        ;          Function 0CH: Flush Buffer, Read Keyboard         ;
        ;                                                            ;
        ;          int flush_kbd()                                   ;
        ;                                                            ;
        ;          Returns 0.                                        ;
        ;                                                            ;
        ;************************************************************;

cProc   flush_kbd,PUBLIC
cBegin
        mov     ax,0c00h        ; Just flush type-ahead buffer.
        int     21h             ; Call MS-DOS.
        xor     ax,ax           ; Return 0.
cEnd
```

Interrupt 21H (33) Function 0DH (13)

1.0 and later

Disk Reset

Function 0DH writes to disk all internal MS-DOS file buffers in memory that have been modified since the last write. All buffers are then marked as "free."

To Call

AH = 0DH

Returns

Nothing

Programmer's Notes

- Function 0DH ensures that the information stored on disk matches changes made by write requests to file buffers in memory.
- Function 0DH does not update the disk directory. The application must issue Function 10H (Close File with FCB) or Function 3EH (Close File) to update directory information correctly.
- Function 0DH should be part of Control-C interrupt-handling routines so that the system is left in a known state when an application is terminated.
- Disk Reset calls can be issued after particularly important disk write calls, such as transactions in an accounting application. Repeated use of this function, however, degrades system performance by defeating the MS-DOS buffering scheme.

Related Functions

10H (Close File with FCB)
3EH (Close File)

Example

```
;************************************************************;
;                                                            ;
;                  Function 0DH: Disk Reset                  ;
;                                                            ;
;                  int reset_disk()                          ;
;                                                            ;
;                  Returns 0.                                ;
;                                                            ;
;************************************************************;
```

(more)

```
cProc   reset_disk,PUBLIC
cBegin
        mov     ah,0dh          ; Set function code.
        int     21h             ; Ask MS-DOS to write all dirty file
                                ; buffers to the disk.
        xor     ax,ax           ; Return 0.
cEnd
```

Interrupt 21H (33) Function 0EH (14)

1.0 and later

Select Disk

Function 0EH sets the default disk drive to the drive specified in the DL register. The default is the disk drive MS-DOS chooses for file access when a filename is specified without a drive designator. A successful call to this function returns the number of logical (not physical) drives in the system.

To Call

AH = 0EH
DL = drive number (0 = drive A, 1 = drive B, 2 = drive C, and so on)

Returns

AL = number of logical drives in the system

Programmer's Notes

- The value used as a drive number is the ASCII value of the uppercase drive letter minus the ASCII value of the uppercase letter A (41H); thus, 0 = drive A, 1 = drive B, and so on.
- A logical drive is defined as any block-oriented device; this category includes floppy-disk drives, RAMdisks, tape devices, fixed disks (which can be partitioned into more than one logical drive), and network drives.
- The maximum numbers of drive designators available for each MS-DOS version are as follows:

MS-DOS Version	Number of Designators	Values
1.x	16	0 through 0FH
2.x	63	0 through 3FH
3.x	26	0 through 19H

Drive letters should be limited to A through P (0 through 0FH) to ensure that an application runs on all versions of MS-DOS.

- With versions of MS-DOS earlier than 3.0 running on IBM PCs and compatibles with one floppy-disk drive, Function 0EH returns 02H as the drive count, because the single physical drive is equivalent to the two logical drives A and B. MS-DOS versions 3.0 and later return a minimum value of 05H in AL.
- On IBM PCs and compatibles, the number of physical floppy-disk drives in a system can be obtained from the ROM BIOS with Interrupt 11H (Equipment Determination).

Related Function

19H (Get Current Disk)

Example

```
        ;*********************************************************;
        ;                                                         ;
        ;     Function 0EH: Select Disk                           ;
        ;                                                         ;
        ;     int select_drive(drive_ltr)                         ;
        ;         char drive_ltr;                                 ;
        ;                                                         ;
        ;     Returns number of logical drives present in system. ;
        ;                                                         ;
        ;*********************************************************;

cProc   select_drive,PUBLIC
parmB   drive_ltr
cBegin
        mov     dl,drive_ltr     ; Get new drive letter.
        and     dl,not 20h      ; Make sure letter is uppercase.
        sub     dl,'A'          ; Convert drive letter to number,
                                ; 'A' = 0, 'B' = 1, etc.
        mov     ah,0eh          ; Set function code.
        int     21h             ; Ask MS-DOS to set default drive.
        cbw                     ; Clear high byte of return value.
cEnd
```

Interrupt 21H (33) Function 0FH (15)

1.0 and later

Open File with FCB

Function 0FH opens the file named in the file control block (FCB) pointed to by DS:DX.

To Call

AH = 0FH
DS:DX = segment:offset of an unopened FCB

Returns

If function is successful:

AL = 00H

If function is not successful:

AL = FFH

Programmer's Notes

- MS-DOS provides several types of file services: FCB file services, which are relatively compatible with the CP/M methods of file handling; extended FCB file services, which take advantage of both CP/M compatibility and MS-DOS extensions; and handle, or "stream-oriented," file services, which are more compatible with UNIX/XENIX and support pathnames (MS-DOS versions 2.0 and later).
- Function 0FH does not support pathnames and so is capable of opening files only in the current directory of the specified drive.
- Function 0FH does not create a new file if the specified file does not already exist. Function 16H (Create File with FCB) is used to create new files with FCBs.
- Function 0FH must use an unopened FCB—that is, one in which all but the drive-designator, filename, and extension fields are zero. If the call is successful, the function fills in the file size and date fields from the file's directory entry. In MS-DOS versions 2.0 and later, the function also fills in the time field.
- If the file is opened on the default drive (the drive number in the FCB is set to 0), MS-DOS fills in the actual drive code. Thus, at some later point in processing, the default drive can be changed and MS-DOS will still have the drive number in the FCB for use in accessing the file. It will therefore continue to use the correct drive.
- If Function 0FH is successful, MS-DOS sets the current-block field to 0; that is, the file pointer is at the beginning of the file. It also sets the record size to 128 bytes (the system default).
- If a record size other than 128 is needed, the record size field of the FCB should be changed after the file is successfully opened and before attempting any I/O.

- In a network running under MS-DOS version 3.1 or later, files are opened by Function 0FH with the share code set to compatibility mode and the access code set to read/write.
- If Function 0FH returns an error code (0FFH) in the AL register, the attempt to open the file was not successful. Possible causes for the failure are
  - File was not found.
  - File has the hidden or system attribute and a properly formatted extended FCB was not used.
  - Filename was improperly specified in the FCB.
  - SHARE is loaded and the file is already open by another process in a mode other than compatibility mode.
- With MS-DOS versions 3.0 and later, Function 59H (Get Extended Error Information) can be used to determine why the attempt to open the file failed.
- MS-DOS passes the first two command-tail parameters into default FCBs located at offsets 5CH and 6CH in the program segment prefix (PSP). Many applications designed to run as .COM files take advantage of one or both of these default FCBs.
- With MS-DOS versions 2.0 and later, Function 3DH (Open File with Handle) should be used in preference to Function 0FH.

Related Functions

10H (Close File with FCB)
16H (Create File with FCB)
3CH (Create File with Handle)
3DH (Open File with Handle)
3EH (Close File)
59H (Get Extended Error Information)
5AH (Create Temporary File)
5BH (Create New File)

Example

```
;*********************************************************;
;                                                         ;
;       Function 0FH: Open File, FCB-based                ;
;                                                         ;
;       int FCB_open(uXFCB,recsize)                       ;
;           char *uXFCB;                                  ;
;           int recsize;                                  ;
;                                                         ;
;       Returns 0 if file opened OK, otherwise returns -1.;
;                                                         ;
;       Note: uXFCB must have the drive and filename      ;
;       fields (bytes 07H through 12H) and the extension  ;
;       flag (byte 00H) set before the call to FCB_open   ;
;       (see Function 29H).                               ;
;                                                         ;
;*********************************************************;
```

(more)

```
cProc   FCB_open,PUBLIC,ds
parmDP  puXFCB
parmW   recsize
cBegin
        loadDP  ds,dx,puXFCB    ; Pointer to unopened extended FCB.
        mov     ah,0fh          ; Ask MS-DOS to open an existing file.
        int     21h
        add     dx,7            ; Advance pointer to start of regular
                                ; FCB.
        mov     bx,dx           ; BX = FCB pointer.
        mov     dx,recsize      ; Get record size parameter.
        mov     [bx+0eh],dx     ; Store record size in FCB.
        xor     dx,dx
        mov     [bx+20h],dl     ; Set current-record
        mov     [bx+21h],dx     ; and relative-record
        mov     [bx+23h],dx     ; fields to 0.
        cbw                     ; Set return value to 0 or -1.
cEnd
```

Interrupt 21H (33) Function 10H (16)

1.0 and later

Close File with FCB

Function 10H flushes file-related information to disk, closes the file named in the file control block (FCB) pointed to by DS:DX, and updates the file's directory entry.

To Call

AH = 10H
DS:DX = segment:offset of previously opened FCB

Returns

If function is successful:

AL = 00H

If function is not successful:

AL = FFH

Programmer's Notes

- A successful call to Function 10H flushes to disk all MS-DOS internal buffers associated with the file and updates the directory entry and file allocation table (FAT). The function thus ensures that correct information is contained in the copy of the file on disk.
- Because MS-DOS versions 1.x and 2.x do not always detect a disk change, an error can occur if the user changes disks between the time the file is opened and the time it is closed. In the worst case, the FAT and the directory of the newly inserted disk may be damaged.
- With MS-DOS versions 2.0 and later, Function 3EH (Close File) should be used in preference to Function 10H.

Related Functions

0FH (Open File with FCB)
3EH (Close File)

Example

```
;*************************************************************;
;                                                             ;
;          Function 10H: Close file, FCB-based                ;
;                                                             ;
;             int FCB_close(oXFCB)                            ;
;                 char *oXFCB;                                ;
;                                                             ;
;             Returns 0 if file closed OK, otherwise          ;
;             returns -1.                                     ;
;                                                             ;
;*************************************************************;

cProc   FCB_close,PUBLIC,ds
parmDP  poXFCB
cBegin
        loadDP  ds,dx,poXFCB    ; Pointer to opened extended FCB.
        mov     ah,10h          ; Ask MS-DOS to close file.
        int     21h
        cbw                     ; Set return value to 0 or -1.
cEnd
```

Interrupt 21H (33) Function 11H (17)

1.0 and later

Find First File

Function 11H searches the current directory for the first file that matches a specified name and extension.

To Call

AH = 11H
DS:DX = segment:offset of unopened file control block (FCB)

Returns

If function is successful:

AL = 00H

Disk transfer area (DTA) contains unopened FCB of same type (normal or extended) as search FCB.

If function is not successful:

AL = FFH

Programmer's Notes

- If necessary, Function 1AH (Set DTA Address) should be used before Function 11H is called, to set the location of the DTA in which the results of the search will be placed.
- With MS-DOS versions 1.0 and later, the wildcard character ? is allowed in the filename. With MS-DOS versions 3.0 and later, both wildcard characters (? and *) are allowed in filenames. Pathnames are not supported.
- With MS-DOS versions 2.0 and later, the attribute field of an extended FCB can be used to search for files with the hidden, system, subdirectory, or volume-label attributes. In such a search, specifying either the normal (00H) or volume-label (08H) attribute restricts MS-DOS to files with the given attribute. Specifying any combination of the hidden (02H), system (04H), and subdirectory (10H) attributes, however, causes MS-DOS to search both for normal files and for those that match the specified attributes.
- For a normal FCB, Function 11H places the drive number in the first byte of the DTA and fills the succeeding 32 bytes with the directory entry.

  For an extended FCB, Function 11H fills in the first 7 bytes of the DTA as follows: the first byte contains 0FFH, indicating an extended FCB; the second through sixth bytes contain 00H, as required by MS-DOS; the seventh byte contains the value of the attribute byte in the search FCB. The next 33 bytes contain the drive number and directory information, as for a normal FCB.

- As with other FCB functions, the number 0 can be used to indicate the default drive. MS-DOS fills in the actual drive number and continues to use that drive for calls to Function 12H (Find Next File) that use the same FCB, regardless of any subsequent selection of a different default drive.
- The FCB with the initial file specifications must remain unmodified if Function 12H is used to continue the search.
- Error reporting in Function 11H is incomplete. An error return (0FFH in the AL register) does not always mean that the file does not exist. Other possibilities include
  - Filename in the FCB was improperly specified.
  - If an extended FCB was used, no files match the attributes given.

  With MS-DOS versions 3.0 and later, Function 59H (Get Extended Error Information) can be used to obtain additional information about the error.
- With MS-DOS versions 2.0 and later, Functions 4EH (Find First File) and 4FH (Find Next File) should be used in preference to Functions 11H and 12H.

Related Functions

12H (Find Next File)
1AH (Set DTA Address)
4EH (Find First File)
4FH (Find Next File)

Example

```
;*************************************************************;
;                                                             ;
;       Function 11H: Find First File, FCB-based              ;
;                                                             ;
;       int FCB_first(puXFCB,attrib)                          ;
;           char *puXFCB;                                     ;
;           char  attrib;                                     ;
;                                                             ;
;       Returns 0 if match found, otherwise returns -1.       ;
;                                                             ;
;       Note: The FCB must have the drive and                 ;
;       filename fields (bytes 07H through 12H) and           ;
;       the extension flag (byte 00H) set before              ;
;       the call to FCB_first (see Function 29H).             ;
;                                                             ;
;*************************************************************;
```

(more)

```
cProc   FCB_first,PUBLIC,ds
parmDP  puXFCB
parmB   attrib
cBegin
        loadDP  ds,dx,puXFCB    ; Pointer to unopened extended FCB.
        mov     bx,dx           ; BX points at FCB, too.
        mov     al,attrib       ; Get search attribute.
        mov     [bx+6],al       ; Put attribute into extended FCB
                                ; area.
        mov     byte ptr [bx],0ffh ; Set flag for extended FCB.
        mov     ah,11h          ; Ask MS-DOS to find 1st matching
                                ; file in current directory.
        int     21h             ; If match found, directory entry can
                                ; be found at DTA address.
        cbw                     ; Set return value to 0 or -1.
cEnd
```

Interrupt 21H (33) Function 12H (18)

1.0 and later

Find Next File

Function 12H searches the current directory for the next file that matches a specified filename and extension. The function assumes a previous successful call to Function 11H (Find First File) with the same file control block (FCB).

To Call

AH = 12H
DS:DX = segment:offset of search FCB

Returns

If function is successful:

AL = 00H

Disk transfer area (DTA) contains unopened FCB of same type (normal or extended) as search FCB.

If function is not successful:

AL = FFH

Programmer's Notes

- Function 12H assumes that a successful call to Function 11H (Find First File) has been completed with the same FCB. The FCB specifies the search pattern. This function also assumes that the wildcard character ? appears at least once in the filename or extension specified.
- An error (indicated by 0FFH returned in register AL) does not necessarily mean that a file matching the file specification does not exist in the current directory. MS-DOS relies on certain information that appears in the search FCB initialized by Function 11H, so it is important not to alter that FCB either between calls to Functions 11H and 12H or between subsequent calls to Function 12H.
- If drive code 0 (the default drive) was used in the call to Function 11H, MS-DOS has already filled in the actual drive number for the current directory. MS-DOS continues to use that drive for all calls to Function 12H that use the same FCB, regardless of the default drive in effect at the time of the call.
- With MS-DOS versions 2.0 and later, Functions 4EH (Find First File) and 4FH (Find Next File) should be used in preference to Functions 11H and 12H.

Related Functions

11H (Find First File)
1AH (Set DTA Address)
4EH (Find First File)
4FH (Find Next File)

Example

```
        ;*********************************************************;
        ;                                                         ;
        ;       Function 12H: Find Next File, FCB-based           ;
        ;                                                         ;
        ;       int FCB_next(puXFCB)                              ;
        ;           char *puXFCB;                                 ;
        ;                                                         ;
        ;       Returns 0 if match found, otherwise returns -1.   ;
        ;                                                         ;
        ;       Note: The FCB must have the drive and             ;
        ;       filename fields (bytes 07H through 12H) and       ;
        ;       the extension flag (byte 00H) set before          ;
        ;       the call to FCB_next (see Function 29H).          ;
        ;                                                         ;
        ;*********************************************************;

cProc   FCB_next,PUBLIC,ds
parmDP  puXFCB
cBegin
        loadDP  ds,dx,puXFCB    ; Pointer to unopened extended FCB.
        mov     ah,12h          ; Ask MS-DOS to find next matching
                                ; file in current directory.
        int     21h             ; If match found, directory entry can
                                ; be found at DTA address.
        cbw                     ; Set return value to 0 or -1.
cEnd
```

Interrupt 21H (33) Function 13H (19)

1.0 and later

Delete File

Function 13H deletes all files matching a specified name and extension from the current directory.

To Call

AH = 13H
DS:DX = segment:offset of an unopened file control block (FCB)

Returns

If function is successful:

AL = 00H

If function is not successful:

AL = FFH

Programmer's Notes

- The wildcard character ? can be used to match any character or sequence of characters in specifying the filename and extension.
- Open files must not be deleted.
- Function 13H does not support pathnames.
- An error (indicated by 0FFH returned in register AL) does not necessarily mean that the filename specified does not exist in the current directory. Other possible causes for an error include
  - Filename in the FCB is improperly specified.
  - File is a read-only, hidden, or system file and an extended FCB with the appropriate attribute byte was not used.
  - Program attempted to delete a volume label and the label does not exist or a properly formatted extended FCB was not used.
  - In networking environments, file is locked or access rights are insufficient for deletion.
- MS-DOS removes file allocation table (FAT) mapping for the file or files deleted by this function and flushes the FAT to disk to ensure that the disk contains a correct table. The first character of the filename in the directory entry is replaced by the value 0E5H, indicating a deleted file.
- Because the function does not physically erase data, use of Function 13H alone is not sufficient in security-critical applications that strictly prohibit viewing the data.

- On networks running under MS-DOS versions 3.1 and later, the user must have Create access rights to the directory containing the file to be deleted.
- Because Function 13H deletes all files matching a given file specification, a conservative approach is to use a combination of Functions 11H (Find First File) and 12H (Find Next File) to build a list of files matching the file specification and then obtain confirmation from the user before deleting the files in the list.
- With MS-DOS versions 2.0 and later, Function 41H (Delete File) should be used in preference to Function 13H.

Related Function

41H (Delete File)

Example

```
        ;*************************************************************;
        ;                                                             ;
        ;       Function 13H: Delete File(s), FCB-based               ;
        ;                                                             ;
        ;       int FCB_delete(uXFCB)                                 ;
        ;           char *uXFCB;                                      ;
        ;                                                             ;
        ;       Returns 0 if file(s) were deleted OK, otherwise       ;
        ;       returns -1.                                           ;
        ;                                                             ;
        ;       Note: uXFCB must have the drive and                   ;
        ;       filename fields (bytes 07H through 12H) and           ;
        ;       the extension flag (byte 00H) set before              ;
        ;       the call to FCB_delete (see Function 29H).            ;
        ;                                                             ;
        ;*************************************************************;

cProc   FCB_delete,PUBLIC,ds
parmDP  puXFCB
cBegin
        loadDP  ds,dx,puXFCB    ; Pointer to unopened extended FCB.
        mov     ah,13h          ; Ask MS-DOS to delete file(s).
        int     21h
        cbw                     ; Return value of 0 or -1.
cEnd
```

Interrupt 21H (33) Function 14H (20)

1.0 and later

Sequential Read

Function 14H reads the next sequential block of data from a file and places the data in the current disk transfer area (DTA).

To Call

AH	= 14H
DS:DX	= segment:offset of a previously opened file control block (FCB)

Returns

AL	= 00H	read successful
	01H	end of file encountered; no data in record
	02H	DTA too small (segment wrap error); read canceled
	03H	end of file; partial record read

If AL = 00H or 03H:

DTA contains data read from file.

Programmer's Notes

- If necessary, Function 1AH (Set DTA Address) should be used to set the base address of the DTA before Function 14H is called. The default DTA is 128 bytes and is located at offset 80H of the program segment prefix (PSP). If record sizes larger than 128 bytes will be used, the program must change the DTA address to point to a buffer of adequate size.
- The read process begins at the current position in the file. When the read is complete, Function 14H increments the current-block and current-record fields of the FCB.
- The size of the record loaded into the DTA is specified in the record size field of the FCB. The default is 128 bytes, set by Function 0FH (Open File with FCB) or Function 16H (Create File with FCB). If the record size is not 128 bytes, the application must set the record size correctly before issuing any reads.
- Function 0FH does not fill in the current-record field of the FCB when opening a file, so this field must be explicitly set (usually to zero) before the first call to Function 14H. The record pointer, which includes the current-block and current-record fields of the FCB, is incremented when Function 14H is successfully completed.
- Function 14H deals with fixed-length records only. Buffering logic must be added to an application if variable-length records are to be manipulated.
- The block of data to be read can be chosen by changing the current-block and current-record fields of the FCB.

- Partial records read at the end of a file are padded with zeros to the requested record length.
- On networks running under MS-DOS version 3.1 or later, the user must have Read access rights to the directory containing the file to be read.
- With MS-DOS versions 2.0 and later, Function 3FH (Read File or Device) should be used in preference to Function 14H.

Related Functions

15H (Sequential Write)
1AH (Set DTA Address)
21H (Random Read)
27H (Random Block Read)
3FH (Read File or Device)

Example

```
        ;************************************************************;
        ;                                                            ;
        ;           Function 14H: Sequential Read, FCB-based         ;
        ;                                                            ;
        ;           int FCB_sread(oXFCB)                             ;
        ;               char *oXFCB;                                 ;
        ;                                                            ;
        ;           Returns 0 if record read OK, otherwise           ;
        ;           returns error code 1, 2, or 3.                   ;
        ;                                                            ;
        ;************************************************************;

cProc   FCB_sread,PUBLIC,ds
parmDP  poXFCB
cBegin
        loadDP  ds,dx,poXFCB    ; Pointer to opened extended FCB.
        mov     ah,14h          ; Ask MS-DOS to read next record,
                                ; placing it at DTA.
        int     21h
        cbw                     ; Clear high byte for return value.
cEnd
```

Interrupt 21H (33) Function 15H (21)

1.0 and later

Sequential Write

Function 15H writes the next sequential block of data from the disk transfer area (DTA) to a specified file.

To Call

AH = 15H
DS:DX = segment:offset of a previously opened file control block (FCB)

DTA contains data to write.

Returns

AL	= 00H	block written successfully
	01H	disk full; write canceled
	02H	DTA too small (segment wrap error); write canceled

Programmer's Notes

- If necessary, the calling process should set the DTA address with Function 1AH (Set DTA Address) to point to the data to be written before issuing a call to Function 15H. The default address of the DTA is offset 80H in the program segment prefix (PSP).
- The FCB must already have been filled in by a call to Function 0FH (Open File with FCB) before Function 15H is called.
- The location of the block to be written is given by the current-block and current-record fields of the FCB. If the write is successful, Function 15H increments the current-block and current-record fields.
- The size of the record written by Function 15H is determined by the value in the record size field of the FCB. The default value is 128, set by Function 0FH (Open File with FCB) or Function 16H (Create File with FCB). A process must set the record size in the FCB correctly before issuing any writes.
- Function 15H deals with fixed-length records only. Buffering logic must be added to an application if variable-length records are to be manipulated.
- Function 15H performs a logical, but not necessarily physical, write operation. If less than one sector is being written, MS-DOS moves the record from the DTA to an appropriate MS-DOS internal buffer. When a full sector of data has been buffered, MS-DOS flushes the buffer to disk. Function 0DH (Disk Reset) or Function 10H (Close File with FCB) can be used to flush data to disk before a full sector is buffered.
- On networks running under MS-DOS versions 3.1 and later, the user must have Write access to the directory containing the file to be written to.
- With MS-DOS versions 2.0 and later, Function 40H (Write File or Device) should be used in preference to Function 15H.

Related Functions

14H (Sequential Read)
1AH (Set DTA Address)
22H (Random Write)
28H (Random Block Write)
40H (Write File or Device)

Example

```
;************************************************************;
;                                                            ;
;          Function 15H: Sequential Write, FCB-based         ;
;                                                            ;
;          int FCB_swrite(oXFCB)                             ;
;              char *oXFCB;                                  ;
;                                                            ;
;          Returns 0 if record read OK, otherwise            ;
;          returns error code 1 or 2.                        ;
;                                                            ;
;************************************************************;

cProc   FCB_swrite,PUBLIC,ds
parmDP  poXFCB
cBegin
        loadDP  ds,dx,poXFCB    ; Pointer to opened extended FCB.
        mov     ah,15h          ; Ask MS-DOS to write next record
                                ; from DTA to disk file.
        int     21h
        cbw                     ; Clear high byte for return value.
cEnd
```

Interrupt 21H (33) Function 16H (22)

1.0 and later

Create File with FCB

Function 16H creates a directory entry in the current directory for a specified file and opens the file for use. If the file already exists, it is opened and truncated to zero length.

To Call

AH = 16H
DS:DX = segment:offset of an unopened file control block (FCB)

Returns

If function is successful:

AL = 00H

If function is not successful:

AL = FFH

Programmer's Notes

- Before creating a new directory entry for the specified file, Function 16H searches the current directory for a matching filename. If a match is found, the existing file is opened, but its length is set to 0. In effect, this action erases an existing file and replaces it with a new, empty file of the same name.

  If a matching filename is not found and the directory has room for a new entry, the file is created and opened, and its length is set to 0.
- An extended file control block (FCB) can be used to create a file with a special attribute, such as hidden. Before the Create File call is issued, the attribute byte must be set appropriately.
- A value of 0FFH returned in the AL register can indicate one of several errors:
  - Filename was improperly specified in the FCB.
  - File with the same name exists but is a read-only, hidden, system, or (in MS-DOS versions 3.x and networks) locked file.
  - Disk is full.
  - Current working directory is the root directory, and it is full.
  - User does not have the appropriate access rights to create a file in this directory (in MS-DOS versions 3.x and networks).

  With MS-DOS versions 3.0 and later, Function 59H (Get Extended Error Information) can be used to obtain additional information about an error.
- Upon successful completion of Function 16H, MS-DOS has
  - Created and opened the file specified in the FCB.

- Filled in the date and time fields of the FCB with the current date and time.
- Set file size to zero.

All other changes made to the FCB are similar to those made by Function 0FH (Open File with FCB).

- Pathnames and wildcard characters (? and *) are not supported by Function 16H.
- With MS-DOS versions 2.0 and later, Function 16H has been superseded by Functions 3CH (Create File with Handle), 5AH (Create Temporary File), and 5BH (Create New File).

Related Functions

0FH (Open File with FCB)
3CH (Create File with Handle)
3DH (Open File with Handle)
5AH (Create Temporary File)
5BH (Create New File)

Example

```
        ;************************************************************;
        ;                                                            ;
        ;       Function 16H: Create File, FCB-based                 ;
        ;                                                            ;
        ;       int FCB_create(uXFCB,recsize)                        ;
        ;           char *uXFCB;                                     ;
        ;           int recsize;                                     ;
        ;                                                            ;
        ;       Returns 0 if file created OK, otherwise              ;
        ;       returns -1.                                          ;
        ;                                                            ;
        ;       Note: uXFCB must have the drive and filename         ;
        ;       fields (bytes 07H through 12H) and the               ;
        ;       extension flag (byte 00H) set before the             ;
        ;       call to FCB_create (see Function 29H).               ;
        ;                                                            ;
        ;************************************************************;

cProc   FCB_create,PUBLIC,ds
parmDP  puXFCB
parmW   recsize
cBegin
        loadDP  ds,dx,puXFCB    ; Pointer to unopened extended FCB.
        mov     ah,16h          ; Ask MS-DOS to create file.
        int     21h
        add     dx,7            ; Advance pointer to start of regular
                                ; FCB.
        mov     bx,dx           ; BX = FCB pointer.
        mov     dx,recsize      ; Get record size parameter.
        mov     [bx+0eh],dx     ; Store record size in FCB.
        xor     dx,dx
        mov     [bx+20h],dl     ; Set current-record
        mov     [bx+21h],dx     ; and relative-record
        mov     [bx+23h],dx     ; fields to 0.
        cbw                     ; Set return value to 0 or -1.
cEnd
```

Interrupt 21H (33) Function 17H (23)

1.0 and later

Rename File

Function 17H renames one or more files in the current directory.

To Call

AH = 17H

DS:DX = segment:offset of modified file control block (FCB) in the following nonstandard format:

Byte(s)	Contents
00H	Drive number
01–08H	Old filename (padded with blanks, if necessary)
09–0BH	Old file extension (padded with blanks, if necessary)
0CH–10H	Zeroed out
11H–18H	New filename (padded with blanks, if necessary)
19H–1BH	New file extension (padded with blanks, if necessary)
11CH–24H	Zeroed out

Returns

If function is successful:

AL = 00H

If function is not successful:

AL = FFH

Programmer's Notes

- The wildcard character ? can be used in specifying both the old and the new filenames, but its meaning differs in each case. A wildcard character in the old filename matches any single character or sequence of characters in the directory entry. A wildcard character in the new filename, however, indicates that the corresponding character or characters in the original filename are not to change.
- With MS-DOS versions 2.0 and later, Function 17H views subdirectory entries as files. These subdirectory entries can be renamed using this function and an extended FCB with the appropriate attribute byte.
- A value of 0FFH returned in the AL register can indicate one of several errors:
  - Old filename is improperly specified in the FCB.
  - File with the new filename already exists in the current directory.

- Old file is a read-only file.
- With MS-DOS versions 3.1 and later in a networking environment, the user has insufficient access rights to the directory.

With MS-DOS versions 3.0 and later, Function 59H (Get Extended Error Information) can be used to obtain additional information about the cause of an error.

- With MS-DOS versions 2.0 and later, Function 56H (Rename File) should be used in preference to Function 17H.

Related Function

56H (Rename File)

Example

```
        ;*********************************************************;
        ;                                                         ;
        ;       Function 17H: Rename File(s), FCB-based           ;
        ;                                                         ;
        ;       int FCB_rename(uXFCBold,uXFCBnew)                 ;
        ;           char *uXFCBold,*uXFCBnew;                     ;
        ;                                                         ;
        ;       Returns 0 if file(s) renamed OK, otherwise        ;
        ;       returns -1.                                       ;
        ;                                                         ;
        ;       Note: Both uXFCB's must have the drive and        ;
        ;       filename fields (bytes 07H through 12H) and       ;
        ;       the extension flag (byte 00H) set before          ;
        ;       the call to FCB_rename (see Function 29H).        ;
        ;                                                         ;
        ;*********************************************************;

cProc   FCB_rename,PUBLIC,<ds,si,di>
parmDP  puXFCBold
parmDP  puXFCBnew
cBegin
        loadDP  es,di,puXFCBold ; ES:DI = Pointer to uXFCBold.
        mov     dx,di           ; Save offset in DX.
        add     di,7            ; Advance pointer to start of regular
                                ; FCBold.
        loadDP  ds,si,puXFCBnew ; DS:SI = Pointer to uXFCBnew.
        add     si,8            ; Advance pointer to filename field
                                ; FCBnew.
                                ; Copy name from FCBnew into FCBold
                                ; at offset 11H:
        add     di,11h          ; DI points 11H bytes into old FCB.
        mov     cx,0bh          ; Copy 0BH bytes, moving new
        rep     movsb           ; name into old FCB.
        push    es              ; Set DS to segment of FCBold.
        pop     ds
        mov     ah,17h          ; Ask MS-DOS to rename old
        int     21h             ; file(s) to new name(s).
        cbw                     ; Set return flag to 0 or -1.
cEnd
```

Interrupt 21H (33) Function 19H (25)

1.0 and later

Get Current Disk

Function 19H returns the code for the current disk drive.

To Call

AH = 19H

Returns

AL = drive code (0 = drive A, 1 = drive B, 2 = drive C, and so on)

Programmer's Note

- The drive code returned by Function 19H is zero-based, meaning that drive A = 0, drive B = 1, and so on. This value is unlike the drive code used in file control blocks (FCBs) and in some other MS-DOS functions, such as 1CH (Get Drive Data) and 36H (Get Disk Free Space), in which 0 indicates the default rather than the current drive.

Related Function

0EH (Select Disk)

Example

```
        ;*********************************************************;
        ;                                                         ;
        ;           Function 19H: Get Current Disk                ;
        ;                                                         ;
        ;           int cur_drive()                               ;
        ;                                                         ;
        ;           Returns letter of current "logged" disk.      ;
        ;                                                         ;
        ;*********************************************************;

cProc   cur_drive,PUBLIC
cBegin
        mov     ah,19h          ; Set function code.
        int     21h             ; Get number of logged disk.
        add     al,'A'          ; Convert number to letter.
        cbw                     ; Clear the high byte of return value.
cEnd
```

Interrupt 21H (33) Function 1AH (26)

1.0 and later

Set DTA Address

Function 1AH specifies the location of the disk transfer area (DTA) to be used for file control block (FCB) disk I/O operations.

To Call

AH = 1AH
DS:DX = segment:offset of DTA

Returns

Nothing

Programmer's Notes

- If an application does not specify a disk transfer area, MS-DOS uses a default buffer at offset 80H in the program segment prefix (PSP).
- The DTA specified must be large enough to accommodate the amount of data to be transferred in a single block. The default record size for FCB file operations is 128 bytes; this value can be changed after a file is successfully opened or created by altering the record size field in the FCB. If the DTA is too small for the record size used by the program, other code or data may be damaged.
- The location of the DTA must be far enough from the top of the segment that contains it to avoid errors caused by segment wrap (data wrapping from the end of the segment to the beginning), which will cause the disk transfer to be terminated. Thus, for example, if records of 128 bytes are to be read, the highest location acceptable for the DTA is DS:FF80H.
- The DTA is used by all FCB-based read and write functions. In addition, any application using the following functions must also set up a DTA for use as a scratch area in directory searches:
  - 11H (Find First File)
  - 12H (Find Next File)
  - 4EH (Find First File)
  - 4FH (Find Next File)

Related Function

2FH (Get DTA Address)

Example

```
        ;*********************************************************;
        ;                                                         ;
        ;               Function 1AH: Set DTA Address             ;
        ;                                                         ;
        ;               int set_DTA(pDTAbuffer)                   ;
        ;                   char far *pDTAbuffer;                 ;
        ;                                                         ;
        ;               Returns 0.                                ;
        ;                                                         ;
        ;*********************************************************;

cProc   set_DTA,PUBLIC,ds
parmD   pDTAbuffer
cBegin
        lds     dx,pDTAbuffer   ; DS:DX = pointer to buffer.
        mov     ah,1ah          ; Set function code.
        int     21h             ; Ask MS-DOS to change DTA address.
        xor     ax,ax           ; Return 0.
cEnd
```

Interrupt 21H (33) Function 1BH (27)

1.0 and later

Get Default Drive Data

Function 1BH returns information about the disk in the default drive.

To Call

AH = 1BH

Returns

If function is successful:

AL = number of sectors per cluster (allocation unit)
CX = number of bytes per sector
DX = number of clusters
DS:BX = segment:offset of the file allocation table (FAT) identification byte

If function is not successful:

AL = FFH

Programmer's Notes

- If Function 1BH returns 0FFH in the AL register, the current drive was invalid or a disk error occurred. The most likely causes of the latter are
  - Drive door was open.
  - Disk was not ready.
  - Medium was bad.
  - Disk was unformatted.

  If any of these situations arises, MS-DOS issues Interrupt 24H (critical error). If Interrupt 24H has not been revectored to a critical error handler controlled by the program and the user responds *Ignore* to the MS-DOS *Abort, Retry, Ignore?* message, the error code 0FFH is returned to the program. An application should check the AL register for a value of 0FFH before assuming it has information on the default drive.
- Possible values of the FAT ID byte (for IBM-compatible media) are the following:

Value	Medium
0FFH	Double-sided, 8 sectors/track, 40 tracks/side
0FEH	Single-sided, 8 sectors/track, 40 tracks/side
0FDH	Double-sided, 9 sectors/track, 40 tracks/side
0FCH	Single-sided, 9 sectors/track, 40 tracks/side

(more)

Value	Medium
0F9H	Double-sided, 15 sectors/track, 40 tracks/side or double-sided, 9 sectors/track, 80 tracks/side
0F8H	Fixed disk
0F0H	Others

- With MS-DOS versions 1.x, Function 1BH returns a pointer in DS:BX for the actual memory image of the FAT. In MS-DOS versions 2.0 and later, the function returns a pointer in DS:BX for a copy of the FAT identification byte; the contents of memory beyond the identification byte are not necessarily the FAT memory image. If access to the FAT is necessary, Interrupt 25H (Absolute Disk Read) can be used to read it into memory.
- The FAT ID byte is not enough to identify a drive completely in MS-DOS versions 2.0 and later. In these versions of MS-DOS, Function 36H (Get Disk Free Space) should be used in preference to Function 1BH to avoid the ambiguity caused by the FAT identification byte.
- With MS-DOS versions 3.2 and later, additional drive information can be obtained by inspecting the BIOS parameter block (BPB) obtained with Function 44H (IOCTL) Subfunction 0DH (Generic I/O Control for Block Devices) minor code 60H (Get Device Parameters).
- With MS-DOS versions 2.0 and later, Function 1CH (Get Drive Data) provides the same types of information as Function 1BH, but for a disk in a drive other than the default drive.

Related Functions

1CH (Get Drive Data)
36H (Get Disk Free Space)
44H (IOCTL)

Example

See SYSTEM CALLS: INTERRUPT 21H: Function ICH.

Interrupt 21H (33) Function 1CH (28)

2.0 and later

Get Drive Data

Function 1CH returns information about the disk in a specified drive.

To Call

AH	= 1CH
DL	= drive code (0 = default drive, 1 = drive A, 2 = drive B, 3 = drive C, and so on)

Returns

If function is successful:

AL	= number of sectors per cluster (allocation unit)
CX	= number of bytes per sector
DX	= number of clusters
DS:BX	= segment:offset of the file allocation table (FAT) identification byte

If function is not successful:

AL	= FFH

Programmer's Notes

- Function 1CH is not available with MS-DOS versions 1.x.
- If the function returns 0FFH in the AL register, the drive code was invalid or a disk error occurred. The most likely causes of the latter are
  - Drive door was open.
  - Disk was not ready.
  - Medium was bad.
  - Disk was unformatted.

  If any of these situations arises, MS-DOS issues Interrupt 24H (critical error). If Interrupt 24H has not been revectored to a critical error handler controlled by the program and the user responds *Ignore* to the MS-DOS *Abort, Retry, Ignore?* message, the error code 0FFH is returned to the program. An application should check the AL register for a value of 0FFH before assuming it has information on the specified drive.
- Possible values of the FAT ID byte (for IBM-compatible media) are the following:

Value	Medium
0FFH	Double-sided, 8 sectors/track, 40 tracks/side
0FEH	Single-sided, 8 sectors/track, 40 tracks/side

(more)

Value	Medium
0FDH	Double-sided, 9 sectors/track, 40 tracks/side
0FCH	Single-sided, 9 sectors/track, 40 tracks/side
0F9H	Double-sided, 15 sectors/track, 40 tracks/side or double-sided, 9 sectors/track, 80 tracks/side
0F8H	Fixed disk
0F0H	Others

- The contents of memory beyond the identification byte pointed to by DS:BX are not necessarily the FAT memory image. If access to the FAT is necessary, Interrupt 25H (Absolute Disk Read) can be used to read it into memory.
- The FAT ID byte is not enough to identify a drive completely. To avoid the ambiguity caused by the FAT identification byte, Function 36H (Get Disk Free Space) should be used in preference to Function 1CH.
- With MS-DOS versions 3.2 and later, additional drive information can be obtained by inspecting the BIOS parameter block (BPB) obtained with Function 44H (IOCTL) Subfunction 0DH (Generic I/O Control for Block Devices) minor code 60H (Get Device Parameters).

Related Functions

1BH (Get Default Drive Data)
36H (Get Disk Free Space)
44H (IOCTL)

Example

```
;************************************************************;
;                                                            ;
;   Function 1CH: Get Drive Data                             ;
;                                                            ;
;   Get information about the disk in the specified          ;
;   drive.  Set drive_ltr to binary 0 for default drive info. ;
;                                                            ;
;   int get_drive_data(drive_ltr,                            ;
;          pbytes_per_sector,                                ;
;          psectors_per_cluster,                             ;
;          pclusters_per_drive)                              ;
;        int  drive_ltr;                                     ;
;        int *pbytes_per_sector;                             ;
;        int *psectors_per_cluster;                          ;
;        int *pclusters_per_drive;                           ;
;                                                            ;
;   Returns -1 for invalid drive, otherwise returns          ;
;   the disk's type (from the 1st byte of the FAT).          ;
;                                                            ;
;************************************************************;
```

(more)

```
cProc   get_drive_data,PUBLIC,<ds,si>
parmB   drive_ltr
parmDP  pbytes_per_sector
parmDP  psectors_per_cluster
parmDP  pclusters_per_drive
cBegin
        mov     si,ds           ; Save DS in SI to use later.
        mov     dl,drive_ltr    ; Get drive letter.
        or      dl,dl           ; Leave 0 alone.
        jz      gdd
        and     dl,not 20h      ; Convert letter to uppercase.
        sub     dl,'A'-1        ; Convert to drive number: 'A' = 1,
                                ; 'B' = 2, etc.
gdd:
        mov     ah,1ch          ; Set function code.
        int     21h             ; Ask MS-DOS for data.
        cbw                     ; Extend AL into AH.
        cmp     al,0ffh         ; Bad drive letter?
        je      gddx            ; If so, exit with error code -1.
        mov     bl,[bx]         ; Get FAT ID byte from DS:BX.
        mov     ds,si           ; Get back original DS.
        loadDP  ds,si,pbytes_per_sector
        mov     [si],cx         ; Return bytes per sector.
        loadDP  ds,si,psectors_per_cluster
        mov     ah,0
        mov     [si],ax         ; Return sectors per cluster.
        loadDP  ds,si,pclusters_per_drive
        mov     [si],dx         ; Return clusters per drive.
        mov     al,bl           ; Return FAT ID byte.
gddx:
cEnd
```

Interrupt 21H (33)
Function 21H (33)

1.0 and later

Random Read

Function 21H reads a selected record from disk into memory.

To Call

AH	= 21H
DS:DX	= segment:offset of previously opened file control block (FCB)

Returns

AL	= 00H	record read successfully
	01H	end of file; no record read
	02H	DTA too small (segment wrap error); read canceled
	03H	end of file; partial record transferred

If AL = 00H or 03H:

DTA contains data read from file.

Programmer's Notes

- Function 21H reads the record into the current disk transfer area (DTA). Unless the 128-byte default DTA (at offset 80H in the program segment prefix) is adequate, Function 1AH (Set DTA Address) should be used to set the DTA address before Function 21H is called. The program must ensure that the buffer pointed to by the DTA address is large enough to hold the records to be transferred.
- The relative-record field in the FCB must be set to the record number to be read. Numbering begins with record 00H; thus, the value 06H in the relative-record field would indicate the seventh record, not the sixth.
- Function 21H sets the current-block and current-record fields to match the relative-record field before transferring the data to the DTA.
- Unlike Function 27H (Random Block Read), Function 21H does not increment the current-block, current-record, or relative-record fields.
- The record length read is determined by the record size field of the FCB.
- If a partial record is read and the end of file is encountered, the remainder of the record is filled out to the requested length with zero bytes.
- On networks running under MS-DOS version 3.1 or later, the user must have Read access rights to the directory containing the file to be read.
- With MS-DOS versions 2.0 and later, Function 3FH (Read File or Device) should be used in preference to Function 21H.

Related Functions

14H (Sequential Read)
1AH (Set DTA Address)
22H (Random Write)
24H (Set Relative Record)
27H (Random Block Read)
3FH (Read File or Device)

Example

```
        ;*********************************************************;
        ;                                                         ;
        ;          Function 21H: Random File Read, FCB-based      ;
        ;                                                         ;
        ;          int FCB_rread(oXFCB,recnum)                    ;
        ;              char *oXFCB;                               ;
        ;              long recnum;                               ;
        ;                                                         ;
        ;          Returns 0 if record read OK, otherwise         ;
        ;          returns error code 1, 2, or 3.                 ;
        ;                                                         ;
        ;*********************************************************;

cProc   FCB_rread,PUBLIC,ds
parmDP  poXFCB
parmD   recnum
cBegin
        loadDP  ds,dx,poXFCB    ; Pointer to opened extended FCB.
        mov     bx,dx           ; BX points at FCB, too.
        mov     ax,word ptr (recnum)    ; Get low 16 bits of record
        mov     [bx+28h],ax             ; number and store in FCB.
        mov     ax,word ptr (recnum+2)  ; Get high 16 bits of record
        mov     [bx+2ah],ax             ; number and store in FCB.
        mov     ah,21h          ; Ask MS-DOS to read recnum'th
                                ; record, placing it at DTA.
        int     21h
        cbw                     ; Clear high byte of return value.
cEnd
```

Interrupt 21H (33) Function 22H (34)

1.0 and later

Random Write

Function 22H writes data from the current disk transfer area (DTA) to a specified record location in a file.

To Call

AH = 22H
DS:DX = segment:offset of previously opened file control block (FCB)

DTA contains data to write.

Returns

AL	= 00H	record written successfully
	01H	disk full
	02H	DTA too small (segment wrap error); write canceled

Programmer's Notes

- Before calling Function 22H, the program must set the disk transfer area (DTA) address appropriately with a call to Function 1AH (Set DTA Address), if necessary, and place the data to be written in the DTA.
- The relative-record field in the FCB must be set to the record number that is to be written. Numbering begins with record 00H; thus, the value 06H in the relative-record field would indicate the seventh record, not the sixth.
- Function 22H sets the current-block and current-record fields to match the relative-record field before writing the data from the DTA.
- Unlike Function 28H (Random Block Write), Function 22H does not increment the current-block, current-record, or relative-record fields.
- The record size field determines the record length written by the function.
- If a record is written beyond the current end of file, the data between the old end of file and the beginning of the new record is uninitialized.
- The file that is written to cannot have the read-only attribute.
- Information is written logically, but not always physically, to disk at the time Function 22H is called. The contents of the DTA are written immediately to disk only if they constitute a sector's worth of information. If less than a sector is written, it is transferred from the DTA to an MS-DOS buffer and is not physically written to disk until one of the following occurs:
  - A full sector of information is ready.
  - The file is closed.
  - Function 0DH (Disk Reset) is issued.

- On networks running under MS-DOS version 3.1 or later, the user must have Write access rights to the directory containing the file to be written to.
- With MS-DOS versions 2.0 and later, Function 40H (Write File or Device) should be used in preference to Function 22H.

Related Functions

15H (Sequential Write)
1AH (Set DTA Address)
21H (Random Read)
24H (Set Relative Record)
28H (Random Block Write)
40H (Write File or Device)

Example

```
        ;*********************************************************;
        ;                                                         ;
        ;          Function 22H: Random File Write, FCB-based     ;
        ;                                                         ;
        ;          int FCB_rwrite(oXFCB,recnum)                   ;
        ;              char *oXFCB;                               ;
        ;              long recnum;                               ;
        ;                                                         ;
        ;          Returns 0 if record read OK, otherwise         ;
        ;          returns error code 1 or 2.                     ;
        ;                                                         ;
        ;*********************************************************;

cProc   FCB_rwrite,PUBLIC,ds
parmDP  poXFCB
parmD   recnum
cBegin
        loadDP  ds,dx,poXFCB    ; Pointer to opened extended FCB.
        mov     bx,dx           ; BX points at FCB, too.
        mov     ax,word ptr (recnum)    ; Get low 16 bits of record
        mov     [bx+28h],ax             ; number and store in FCB.
        mov     ax,word ptr (recnum+2)  ; Get high 16 bits of record
        mov     [bx+2ah],ax             ; number and store in FCB.
        mov     ah,22h          ; Ask MS-DOS to write DTA to
        int     21h             ; recnum'th record of file.
        cbw                     ; Clear high byte for return value.
cEnd
```

Interrupt 21H (33) Function 23H (35)

1.0 and later

Get File Size

Function 23H searches the current directory for a specified file and returns the size of the file in records.

To Call

AH = 23H
DS:DX = segment:offset of unopened file control block (FCB) with record size field set appropriately

Returns

If function is successful:

AL = 00H

FCB relative-record field contains number of records, rounded upward if necessary.

If function is not successful:

AL = FFH

Programmer's Notes

- The record size field in the FCB can be set to 1 to find the number of bytes in the file.
- The number of records is the file size divided by the record size. If there is a remainder, the record count is rounded upward. The result stored in the relative-record field may, therefore, contain a value that is 1 larger than the number of complete records in the file.
- Because record numbers are zero based and this function returns the number of records in a file in the relative-record field of the FCB, Function 23H can be used to position the file pointer to the end of file.
- With MS-DOS versions 2.0 and later, Function 42H (Move File Pointer) should be used in preference to Function 23H.

Related Function

42H (Move File Pointer)

Example

```
        ;*************************************************************;
        ;                                                             ;
        ;       Function 23H: Get File Size, FCB-based                ;
        ;                                                             ;
        ;       long FCB_nrecs(uXFCB,recsize)                         ;
        ;            char *uXFCB;                                     ;
        ;            int recsize;                                     ;
        ;                                                             ;
        ;       Returns a long -1 if file not found, otherwise        ;
        ;       returns the number of records of size recsize.        ;
        ;                                                             ;
        ;       Note: uXFCB must have the drive and                   ;
        ;       filename fields (bytes 07H through 12H) and           ;
        ;       the extension flag (byte 00H) set before              ;
        ;       the call to FCB_nrecs (see Function 29H).             ;
        ;                                                             ;
        ;*************************************************************;

cProc   FCB_nrecs,PUBLIC,ds
parmDP  puXFCB
parmW   recsize
cBegin
        loadDP  ds,dx,puXFCB    ; Pointer to unopened extended FCB.
        mov     bx,dx           ; Copy FCB pointer into BX.
        mov     ax,recsize      ; Get record size
        mov     [bx+15h],ax     ; and store it in FCB.
        mov     ah,23h          ; Ask MS-DOS for file size (in
                                ; records).
        int     21h
        cbw                     ; If AL = 0FFH, set AX to -1.
        cwd                     ; Extend to long.
        or      dx,dx           ; Is DX negative?
        js      nr_exit         ; If so, exit with error flag.
        mov     [bx+2bh],al     ; Only low 24 bits of the relative-
                                ; record field are used, so clear the
                                ; top 8 bits.
        mov     ax,[bx+28h]     ; Return file length in DX:AX.
        mov     dx,[bx+2ah]
nr_exit:
cEnd
```

Interrupt 21H (33) Function 24H (36)

1.0 and later

Set Relative Record

Function 24H sets the relative-record field of a file control block (FCB) to match the file position indicated by the current-block and current-record fields of the same FCB.

To Call

AH = 24H
DS:DX = segment:offset of previously opened FCB

Returns

AL = 00H

Relative-record field is modified in FCB.

Programmer's Notes

- The AL register is always set to 00H by Function 24H. Thus, any preexisting information in the AL register is lost.
- Before Function 24H is called, the program must open the FCB with Function 0FH (Open File with FCB) or with Function 16H (Create File with FCB).
- The entire relative-record field (4 bytes) of the FCB must be initialized to zeros before calling Function 24H. If this is not done, any value in the high-order byte of the high-order word remaining from previous reads or writes might not be overwritten and the resulting relative-record number will be invalid.
- Function 24H is normally used in changing from sequential to random I/O. Sequential I/O, performed by Functions 14H (Sequential Read) and 15H (Sequential Write), sets the current-block and current-record fields of the FCB. Random I/O uses the relative-record field, which is set by Function 24H to match the current file position as recorded in the current-block and current-record fields.

  After the file pointer is set, any of the following functions can be used to access data at the record pointed to by the relative-record field:
  - 21H (Random Read)
  - 22H (Random Write)
  - 27H (Random Block Read)
  - 28H (Random Block Write)
- With MS-DOS versions 2.0 and later, Function 42H (Move File Pointer) should be used in preference to Function 24H.

Related Function

42H (Move File Pointer)

Example

```
        ;*********************************************************;
        ;                                                         ;
        ;             Function 24H: Set Relative Record           ;
        ;                                                         ;
        ;             int FCB_set_rrec(oXFCB)                     ;
        ;                 char *oXFCB;                            ;
        ;                                                         ;
        ;             Returns 0.                                  ;
        ;                                                         ;
        ;*********************************************************;

cProc   FCB_set_rrec,PUBLIC,ds
parmDP  poXFCB
cBegin
        loadDP  ds,dx,poXFCB    ; Pointer to opened extended FCB.
        mov     bx,dx           ; BX points at FCB, too.
        mov     byte ptr [bx+2bh],0 ; Zero high byte of high word of
                                    ; relative-record field.
        mov     ah,24h          ; Ask MS-DOS to set relative record
                                ; to current record.
        int     21h
        xor     ax,ax           ; Return 0.
cEnd
```

Interrupt 21H (33) Function 25H (37)

1.0 and later

Set Interrupt Vector

Function 25H sets an address in the interrupt vector table to point to a specified interrupt handler.

To Call

AH	= 25H
AL	= interrupt number
DS:DX	= segment:offset of interrupt handler

Returns

Nothing

Programmer's Notes

- When Function 25H is called, the 4-byte address in DS:DX is placed in the correct position in the interrupt vector table.
- Function 25H is the recommended method for initializing or changing an interrupt vector. A vector in the interrupt vector table should never be changed directly.
- Before Function 25H is used to change an interrupt vector, the address of the current interrupt handler should be read with Function 35H (Get Interrupt Vector) and then saved for restoration before the program terminates.

Related Function

35H (Get Interrupt Vector)

Example

```
;*********************************************************;
;                                                         ;
;             Function 25H: Set Interrupt Vector          ;
;                                                         ;
;             typedef void (far *FCP)();                  ;
;             int set_vector(intnum,vector)               ;
;                 int intnum;                             ;
;                 FCP vector;                             ;
;                                                         ;
;             Returns 0.                                  ;
;                                                         ;
;*********************************************************;
```

(more)

```
cProc   set_vector,PUBLIC,ds
parmB   intnum
parmD   vector
cBegin
        lds     dx,vector       ; Get vector segment:offset into
                                ; DS:DX.
        mov     al,intnum       ; Get interrupt number into AL.
        mov     ah,25h          ; Select "set vector" function.
        int     21h             ; Ask MS-DOS to change vector.
        xor     ax,ax           ; Return 0.
cEnd
```

Interrupt 21H (33) Function 26H (38)

1.0 and later

Create New Program Segment Prefix

Function 26H creates a new program segment prefix (PSP) at a specified segment address.

To Call

AH = 26H
DX = segment address of the PSP to create

Returns

Nothing

Programmer's Notes

- Function 26H copies the current PSP to the address indicated by DX. Note that DX contains a segment address, not an absolute address.
- After the copy is made, the memory size information located at offset 06H in the new PSP is adjusted to match the amount of memory available to the new PSP. In addition, the current contents of the interrupt vectors for Interrupt 22H (Terminate Routine Address), Interrupt 23H (Control-C Handler Address), and Interrupt 24H (Critical Error Handler Address) are saved starting at offset 0AH of the new PSP.
- A .COM file can be loaded into memory immediately after the new PSP and execution can begin at that location. A .EXE file cannot be loaded and executed in this manner.
- With MS-DOS versions 2.0 and later, Function 4BH (Load and Execute Program) should be used in preference to Function 26H. Function 4BH can be used to load .COM files, .EXE files, or overlays.

Related Function

4BH (Load and Execute Program)

Example

```
;*************************************************************;
;                                                             ;
;       Function 26H: Create New Program Segment Prefix       ;
;                                                             ;
;       int create_psp(pspseg)                                ;
;           int  pspseg;                                      ;
;                                                             ;
;       Returns 0.                                            ;
;                                                             ;
;*************************************************************;
```

(more)

```
cProc   create_psp,PUBLIC
parmW   pspseg
cBegin
        mov     dx,pspseg       ; Get segment address of new PSP.
        mov     ah,26h          ; Set function code.
        int     21h             ; Ask MS-DOS to create new PSP.
        xor     ax,ax           ; Return 0.
cEnd
```

Interrupt 21H (33) Function 27H (39)

1.0 and later

Random Block Read

Function 27H reads one or more records into memory, placing the records in the current disk transfer area (DTA).

To Call

AH	= 27H
CX	= number of records to read
DS:DX	= segment:offset of previously opened file control block (FCB)

Returns

AL	= 00H	read successful
	01H	end of file; no record read
	02H	DTA too small (segment wrap error); no record read
	03H	end of file; partial record read

If AL is 00H or 03H:

CX = number of records read

DTA contains data read from file.

Programmer's Notes

- The DTA address should be set with Function 1AH (Set DTA Address) before Function 27H is called. If the DTA address has not been set, MS-DOS uses a default 128-byte DTA at offset 80H in the program segment prefix (PSP).
- Function 27H reads the number of records specified in CX sequentially, starting at the file location indicated by the relative-record and record size fields in the FCB. If CX = 0, no records are read.
- The record length used by Function 27H is the value in the record size field of the FCB. Unless a new value is placed in this field after a file is opened or created, MS-DOS uses a default record length of 128 bytes.
- Function 27H is similar to Function 21H (Random Read); however, Function 27H can read more than one record at a time and updates the relative-record field of the FCB after each call. Successive calls to this function thus read sequential groups of records from a file, whereas successive calls to Function 21H repeatedly read the same record.
- Possible alternative causes for end-of-file (01H) errors include
  - Disk removed from drive since file was opened.
  - Previous open failed.

  With MS-DOS versions 3.0 and later, more detailed information on the error can be obtained by calling Function 59H (Get Extended Error Information).

- On networks running under MS-DOS version 3.1 or later, the user must have Read access rights to the directory containing the file to be read.
- With MS-DOS versions 2.0 and later, Function 3FH (Read File or Device) should be used in preference to Function 27H.

Related Functions

14H (Sequential Read)
1AH (Set DTA Address)
21H (Random Read)
24H (Set Relative Record)
28H (Random Block Write)
3FH (Read File or Device)

Example

```
        ;*********************************************************;
        ;                                                         ;
        ;       Function 27H: Random File Block Read, FCB-based   ;
        ;                                                         ;
        ;       int FCB_rblock(oXFCB,nrequest,nactual,start)      ;
        ;           char *oXFCB;                                  ;
        ;           int   nrequest;                               ;
        ;           int  *nactual;                                ;
        ;           long  start;                                  ;
        ;                                                         ;
        ;       Returns read status 0, 1, 2, or 3 and sets        ;
        ;       nactual to number of records actually read.       ;
        ;                                                         ;
        ;       If start is -1, the relative-record field is      ;
        ;       not changed, causing the block to be read starting;
        ;       at the current record.                            ;
        ;                                                         ;
        ;*********************************************************;

cProc   FCB_rblock,PUBLIC,<ds,di>
parmDP  poXFCB
parmW   nrequest
parmDP  pnactual
parmD   start
cBegin
        loadDP  ds,dx,poXFCB    ; Pointer to opened extended FCB.
        mov     di,dx           ; DI points at FCB, too.
        mov     ax,word ptr (start) ; Get long value of start.
        mov     bx,word ptr (start+2)
        mov     cx,ax           ; Is start = -1?
        and     cx,bx
        inc     cx
        jcxz    rb_skip         ; If so, don't change relative-record
                                ; field.
        mov     [di+28h],ax     ; Otherwise, seek to start record.
```

(more)

```
        mov     [di+2ah],bx
rb_skip:
        mov     cx,nrequest     ; CX = number of records to read.
        mov     ah,27h          ; Get MS-DOS to read CX records,
        int     21h             ; placing them at DTA.
        loadDP  ds,bx,pnactual  ; DS:BX = address of nactual.
        mov     [bx],cx         ; Return number of records read.
        cbw                     ; Clear high byte.
cEnd
```

Interrupt 21H (33) Function 28H (40)

1.0 and later

Random Block Write

Function 28H writes one or more records from the current disk transfer area (DTA) to a file.

To Call

AH	= 28H
CX	= number of records to write
DS:DX	= segment:offset of previously opened file control block (FCB)

DTA contains data to write.

Returns

AL	= 00H	write successful
	01H	disk full
	02H	DTA too small (segment wrap error); write canceled

If AL is 00H or 01H:

CX = number of records written

Programmer's Notes

- Data to be written must be placed in the DTA before Function 28H is called. Unless the DTA address has been set with Function 1AH (Set DTA Address), MS-DOS uses a default 128-byte DTA at offset 80H in the program segment prefix (PSP).
- Function 28H writes the number of records indicated in CX, beginning at the location specified in the relative-record field of the file control block (FCB). If Function 28H is called with CX = 0, the file is truncated or extended to the size indicated by the record-size and relative-record fields of the FCB.
- The record length used by Function 28H is the value in the record size field of the FCB. Unless a new value is assigned after a file is opened or created, MS-DOS uses a default record length of 128 bytes.
- Function 28H is similar to Function 22H (Random Write); however, Function 28H can write more than one record at a time and updates the relative-record field of the FCB after each call. Successive calls to this function thus write sequential groups of records to a file, whereas successive calls to Function 22H repeatedly write the same record.

- Possible alternative causes for disk full (01H) errors include
  - Disk removed from drive since file was opened.
  - Previous open failed.

  In MS-DOS versions 3.0 and later, more detailed information on the error can be obtained by calling Function 59H (Get Extended Error Information).
- Information is written logically, but not always physically, to disk at the time Function 28H is called. The contents of the DTA are written immediately to disk only if they constitute a full sector of information. If less than a sector is written, it is transferred from the DTA to an MS-DOS buffer and is not physically written to disk until one of the following occurs:
  - A full sector of information is ready.
  - The file is closed.
  - Function 0DH (Disk Reset) is issued.
- On networks running under MS-DOS version 3.1 or later, the user must have Write access rights to the directory containing the file to be written to.
- With MS-DOS versions 2.0 and later, Function 40H (Write File or Device) should be used in preference to Function 28H.

Related Functions

15H (Sequential Write)
1AH (Set DTA Address)
22H (Random Write)
24H (Set Relative Record)
27H (Random Block Read)
40H (Write File or Device)

Example

```
;*************************************************************;
;                                                             ;
;       Function 28H: Random File Block Write, FCB-based      ;
;                                                             ;
;       int FCB_wblock(oXFCB,nrequest,nactual,start)          ;
;           char *oXFCB;                                      ;
;           int   nrequest;                                   ;
;           int  *nactual;                                    ;
;           long  start;                                      ;
;                                                             ;
;       Returns write status of 0, 1, or 2 and sets           ;
;       nactual to number of records actually written.        ;
;                                                             ;
;       If start is -1, the relative-record field is          ;
;       not changed, causing the block to be written          ;
;       starting at the current record.                       ;
;                                                             ;
;*************************************************************;
```

(more)

```
cProc   FCB_wblock,PUBLIC,<ds,di>
parmDP  poXFCB
parmW   nrequest
parmDP  pnactual
parmD   start
cBegin
        loadDP  ds,dx,poXFCB    ; Pointer to opened extended FCB.
        mov     di,dx           ; DI points at FCB, too.
        mov     ax,word ptr (start) ; Get long value of start.
        mov     bx,word ptr (start+2)
        mov     cx,ax           ; Is start = -1?
        and     cx,bx
        inc     cx
        jcxz    wb_skip         ; If so, don't change relative-record
                                ; field.
        mov     [di+28h],ax     ; Otherwise, seek to start record.
        mov     [di+2ah],bx
wb_skip:
        mov     cx,nrequest     ; CX = number of records to write.
        mov     ah,28h          ; Get MS-DOS to write CX records
        int     21h             ; from DTA to file.
        loadDP  ds,bx,pnactual  ; DS:BX = address of nactual.
        mov     ds:[bx],cx      ; Return number of records written.
        cbw                     ; Clear high byte.
cEnd
```

Interrupt 21H (33) Function 29H (41)

1.0 and later

Parse Filename

Function 29H examines a string for a valid filename in the form *drive:filename.ext*. If the string represents a valid filename, the function creates an unopened file control block (FCB) for it.

To Call

AH = 29H

AL = code to control parsing, as follows (bits 0–3 only):

Bit	Value	Meaning
0	0	Stop parsing if file separator is found.
	1	Ignore leading separators (parse off white space).
1	0	Set drive number field in FCB to 0 (current drive) if string does not include a drive identifier.
	1	Set drive as specified in the string; leave unaltered if string does not include a drive identifier.
2	0	Set filename field in the FCB to blanks (20H) if string does not include a filename.
	1	Leave filename field unaltered if string does not include a filename.
3	0	Set extension field in FCB to blanks (20H) if string does not include a filename extension.
	1	Leave extension field unaltered if string does not include a filename extension.

DS:SI = segment:offset of string to parse

ES:DI = segment:offset of buffer for unopened FCB

Returns

AL = 00H string does not contain wildcard characters

01H string contains wildcard characters

FFH drive specifier invalid

DS:SI = segment:offset of first byte following the parsed string

ES:DI = segment:offset of unopened FCB

Programmer's Notes

- Bits 0 through 3 of the byte in the AL register control the way the text string is parsed; bits 4 through 7 are not used and must be 0.
- After MS-DOS parses the string, DS:SI points to the first byte following the parsed string. If DS:SI points to an earlier byte, MS-DOS did not parse the entire string.
- If Function 29H encounters the MS-DOS wildcard character * (match all remaining characters) in a filename or extension, the remaining bytes in the corresponding FCB field are set to the wildcard character ? (match one character). For example, the string DOS*.D* would be converted to DOS????? in the filename field and D?? in the extension field of the FCB.
- With MS-DOS versions 1.x, the following characters are filename separators:

  : . ; , = + space tab / " [ ]

  With MS-DOS versions 2.0 and later, the following characters are filename separators:

  : . ; , = + space tab

- The following characters are filename terminators:

  / " [ ] < > |
  All filename separators
  Any control character

- If the string does not contain a valid filename, ES:DI+1 points to an ASCII blank character (20H).
- Function 29H cannot parse pathnames.

Related Functions

None

Example

```
;************************************************************;
;                                                            ;
;           Function 29H: Parse Filename into FCB            ;
;                                                            ;
;           int FCB_parse(uXFCB,name,ctrl)                   ;
;               char *uXFCB;                                 ;
;               char *name;                                  ;
;               int ctrl;                                    ;
;                                                            ;
;           Returns -1 if error,                             ;
;                    0 if no wildcards found,                ;
;                    1 if wildcards found.                   ;
;                                                            ;
;************************************************************;
```

(more)

```
cProc   FCB_parse,PUBLIC,<ds,si,di>
parmDP  puXFCB
parmDP  pname
parmB   ctrl
cBegin
        loadDP  es,di,puXFCB    ; Pointer to unopened extended FCB.
        push    di              ; Save DI.
        xor     ax,ax           ; Fill all 22 (decimal) words of the
                                ; extended FCB with zeros.
        cld                     ; Make sure direction flag says UP.
        mov     cx,22d
        rep     stosw
        pop     di              ; Recover DI.
        mov     byte ptr [di],0ffh ; Set flag byte to mark this as an
                                   ; extended FCB.
        add     di,7            ; Advance pointer to start of regular
                                ; FCB.
        loadDP  ds,si,pname     ; Get pointer to filename into DS:SI.
        mov     al,ctrl         ; Get parse control byte.
        mov     ah,29h          ; Parse filename, please.
        int     21h
        cbw                     ; Set return parameter.
cEnd
```

Interrupt 21H (33) Function 2AH (42)

1.0 and later

Get Date

Function 2AH returns the current system date—year, month, day, and day of the week—in binary form.

To Call

AH = 2AH

Returns

AL = day of the week (0 = Sunday, 1 = Monday, 2 = Tuesday, and so on; MS-DOS versions 1.10 and later)
CX = year (1980 through 2099)
DH = month (1 through 12)
DL = day (1 through 31)

Programmer's Note

- Years outside the range 1980–2099 cannot be returned by Function 2AH.

Related Functions

2BH (Set Date)
2CH (Get Time)
2DH (Set Time)

Example

```
;*********************************************************;
;                                                         ;
;           Function 2AH: Get Date                        ;
;                                                         ;
;           long get_date(pdow,pmonth,pday,pyear)         ;
;                char *pdow,*pmonth,*pday;                ;
;                int *pyear;                              ;
;                                                         ;
;           Returns the date packed into a long:          ;
;                low byte  = day of month                 ;
;                next byte = month                        ;
;                next word = year.                        ;
;                                                         ;
;*********************************************************;
```

(more)

```
cProc   get_date,PUBLIC,ds
parmDP  pdow
parmDP  pmonth
parmDP  pday
parmDP  pyear
cBegin
        mov     ah,2ah          ; Set function code.
        int     21h             ; Get date info from MS-DOS.
        loadDP  ds,bx,pdow      ; DS:BX = pointer to dow.
        mov     [bx],al         ; Return dow.
        loadDP  ds,bx,pmonth    ; DS:BX = pointer to month.
        mov     [bx],dh         ; Return month.
        loadDP  ds,bx,pday      ; DS:BX = pointer to day.
        mov     [bx],dl         ; Return day.
        loadDP  ds,bx,pyear     ; DS:BX = pointer to year.
        mov     [bx],cx         ; Return year.
        mov     ax,dx           ; Pack day, month, ...
        mov     dx,cx           ; ... and year into return value.
cEnd
```

Interrupt 21H (33) Function 2BH (43)

1.0 and later

Set Date

Function 2BH accepts binary values for the year, month, and day of the month and stores them in the system's date counter as the number of days since January 1, 1980.

To Call

AH = 2BH
CX = year (1980 through 2099)
DH = month (1 through 12)
DL = day (1 through 31)

Returns

AL = 00H system date updated
FFH invalid date specified

Programmer's Note

- The year must be a 16-bit value in the range 1980 through 2099. Values outside this range are not accepted. In addition, supplying only the last two digits of the year causes an error.

Related Functions

2AH (Get Date)
2CH (Get Time)
2DH (Set Time)

Example

```
;*********************************************************;
;                                                         ;
;            Function 2BH: Set Date                       ;
;                                                         ;
;            int set_date(month,day,year)                 ;
;                char month,day;                          ;
;                int year;                                ;
;                                                         ;
;            Returns 0 if date was OK, -1 if not.         ;
;                                                         ;
;*********************************************************;
```

(more)

```
cProc   set_date,PUBLIC
parmB   month
parmB   day
parmW   year
cBegin
        mov     dh,month        ; Get new month.
        mov     dl,day          ; Get new day.
        mov     cx,year         ; Get new year.
        mov     ah,2bh          ; Set function code.
        int     21h             ; Ask MS-DOS to change date.
        cbw                     ; Return 0 or -1.
cEnd
```

Interrupt 21H (33) Function 2CH (44)

1.0 and later

Get Time

Function 2CH reports the current system time—hours (based on a 24-hour clock), minutes, seconds, and hundredths of a second—in binary form.

To Call

AH = 2CH

Returns

CH = hours (0 through 23)
CL = minutes (0 through 59)
DH = seconds (0 through 59)
DL = hundredths of second (0 through 99)

Programmer's Note

- The accuracy of the time returned by Function 2CH depends on the accuracy of the system's timekeeping hardware. On systems unable to resolve time to the hundredth of a second, the DL register may contain either 00H or an approximate value calculated by an MS-DOS algorithm.

Related Functions

2AH (Get Date)
2BH (Set Date)
2DH (Set Time)

Example

```
;*********************************************************;
;                                                         ;
;            Function 2CH: Get Time                       ;
;                                                         ;
;            long get_time(phour,pmin,psec,phund)         ;
;                 char *phour,*pmin,*psec,*phund;         ;
;                                                         ;
;            Returns the time packed into a long:         ;
;                 low byte  = hundredths                  ;
;                 next byte = seconds                     ;
;                 next byte = minutes                     ;
;                 next byte = hours.                      ;
;                                                         ;
;*********************************************************;
```

(more)

```
cProc   get_time,PUBLIC,ds
parmDP  phour
parmDP  pmin
parmDP  psec
parmDP  phund
cBegin
        mov     ah,2ch          ; Set function code.
        int     21h             ; Get time from MS-DOS.
        loadDP  ds,bx,phour     ; DS:BX = pointer to hour.
        mov     [bx],ch         ; Return hour.
        loadDP  ds,bx,pmin      ; DS:BX = pointer to min.
        mov     [bx],cl         ; Return min.
        loadDP  ds,bx,psec      ; DS:BX = pointer to sec.
        mov     [bx],dh         ; Return sec.
        loadDP  ds,bx,phund     ; DS:BX = pointer to hund.
        mov     [bx],dl         ; Return hund.
        mov     ax,dx           ; Pack seconds, hundredths, ...
        mov     dx,cx           ; ... minutes, and hour into
                                ; return value.
cEnd
```

Interrupt 21H (33) Function 2DH (45)

1.0 and later

Set Time

Function 2DH accepts binary values for the hour (based on a 24-hour clock), minute, second, and hundredths of a second and stores them in the operating system's time counter.

To Call

AH = 2DH
CH = hours (0 through 23)
CL = minutes (0 through 59)
DH = seconds (0 through 59)
DL = hundredths of second (0 through 99)

Returns

AL = 00H time successfully updated
FFH invalid time specified

Programmer's Note

- On systems that are unable to resolve the time to the hundredth of a second, the DL register should be set to 00H before Function 2DH is called.

Related Functions

2AH (Get Date)
2BH (Set Date)
2CH (Get Time)

Example

```
;*********************************************************;
;                                                         ;
;            Function 2DH: Set Time                       ;
;                                                         ;
;            int set_time(hour,min,sec,hund)              ;
;                char hour,min,sec,hund;                  ;
;                                                         ;
;            Returns 0 if time was OK, -1 if not.         ;
;                                                         ;
;*********************************************************;
```

(more)

```
cProc   set_time,PUBLIC
parmB   hour
parmB   min
parmB   sec
parmB   hund
cBegin
        mov     ch,hour         ; Get new hour.
        mov     cl,min          ; Get new minutes.
        mov     dh,sec          ; Get new seconds.
        mov     dl,hund         ; Get new hundredths.
        mov     ah,2dh          ; Set function code.
        int     21h             ; Ask MS-DOS to change time.
        cbw                     ; Return 0 or -1.
cEnd
```

Interrupt 21H (33) Function 2EH (46)

1.0 and later

Set/Reset Verify Flag

Function 2EH turns the internal MS-DOS verify flag on or off, thus determining whether MS-DOS verifies disk write operations.

To Call

AH = 2EH
AL = 00H turn verify off
01H turn verify on
DL = 00H (MS-DOS versions 1.x and 2.x only)

Returns

Nothing

Programmer's Notes

- If the verify flag is on, MS-DOS requests any block-device driver to verify each sector written. If the driver does not support read-after-write verification, the verify flag has no effect.
- Function 54H (Get Verify Flag) can be used to check the current setting of the verify flag.
- Verifying data slows disk access during write operations. Because disk errors are rare, the default setting of the verify flag is off.
- Verification can be controlled at the user level with the MS-DOS VERIFY command.

Related Function

54H (Get Verify Flag)

Example

```
;************************************************************;
;                                                            ;
;            Function 2EH: Set/Reset Verify Flag             ;
;                                                            ;
;            int set_verify(newvflag)                        ;
;                char newvflag;                              ;
;                                                            ;
;            Returns 0.                                      ;
;                                                            ;
;************************************************************;
```

(more)

```
cProc   set_verify,PUBLIC
parmB   newvflag
cBegin
        mov     al,newvflag     ; Get new value of verify flag.
        mov     ah,2eh          ; Set function code.
        int     21h             ; Ask MS-DOS to store flag.
        xor     ax,ax           ; Return 0.
cEnd
```

Interrupt 21H (33) Function 2FH (47)

2.0 and later

Get DTA Address

Function 2FH returns the current disk transfer area (DTA) address.

To Call

AH = 2FH

Returns

ES:BX = segment:offset of current DTA address

Programmer's Notes

- Function 2FH returns the base address of the current DTA. MS-DOS has no way of knowing the size of the buffer at that address; the program must ensure that the buffer pointed to by the DTA address is large enough to hold any records transferred to it.
- The current DTA address can be set with Function 1AH (Set DTA Address). If the DTA address is not set, MS-DOS uses a default buffer of 128 bytes located at offset 80H in the program segment prefix (PSP).

Related Function

1AH (Set DTA Address)

Example

```
;************************************************************;
;                                                            ;
;           Function 2FH: Get DTA Address                    ;
;                                                            ;
;           char far *get_DTA()                              ;
;                                                            ;
;           Returns a far pointer to the DTA buffer.         ;
;                                                            ;
;************************************************************;

cProc   get_DTA,PUBLIC
cBegin
        mov     ah,2fh          ; Set function code.
        int     21h             ; Ask MS-DOS for current DTA address.
        mov     ax,bx           ; Return offset in AX.
        mov     dx,es           ; Return segment in DX.
cEnd
```

Interrupt 21H (33) Function 30H (48)

2.0 and later

Get MS-DOS Version Number

Function 30H returns the major and minor version numbers for MS-DOS versions 2.0 and later.

To Call

AH	= 30H
AL	= 00H

Returns

AL	= major version number (for example, 3 for MS-DOS version 3.x)
AH	= minor version number (for example, 0AH for MS-DOS version x.10)
BH	= original equipment manufacturer's (OEM's) serial number (OEM dependent — usually 00H for PC-DOS, 0FFH or other values for MS-DOS)
BL:CX	= 24-bit user serial number (optional; OEM dependent)

Programmer's Notes

- With MS-DOS versions 1.x, Function 30H returns 00H in the AL register; the value returned in AH is variable and not representative of the actual 1.x minor version number.
- Function 30H supplies the MS-DOS version number to an application program that might require features of the operating system that are not available in all versions. If an application attempts to use such features with the wrong version of MS-DOS, the results are unpredictable.

  Applications requiring MS-DOS version 2.0 or later should use Function 30H to check for versions 1.x. Because versions 1.x do not contain predefined handles for displaying error messages, Function 02H (Character Output) or Function 09H (Display String) must be used with those versions. Similarly, applications running under versions 1.x cannot terminate through a call to Function 4CH (Terminate Process with Return Code).

Related Functions

None

Example

```
;************************************************************;
;                                                            ;
;           Function 30H: Get MS-DOS Version Number          ;
;                                                            ;
;           int DOS_version()                                ;
;                                                            ;
;           Returns number of MS-DOS version, with           ;
;              major version in high byte,                   ;
;              minor version in low byte.                    ;
;                                                            ;
;************************************************************;

cProc   DOS_version,PUBLIC
cBegin
        mov     ax,3000H        ; Set function code and clear AL.
        int     21h             ; Ask MS-DOS for version number.
        xchg    al,ah           ; Swap major and minor numbers.
cEnd
```

Interrupt 21H (33) Function 31H (49)

2.0 and later

Terminate and Stay Resident

Function 31H terminates a program and returns control to the parent process (usually COMMAND.COM) but keeps the terminated program resident in memory.

To Call

AH = 31H
AL = return code
DX = number of paragraphs of memory to be reserved for current process

Returns

Nothing

Programmer's Notes

- The following interrupt vectors are restored from the program segment prefix (PSP) of the terminated program:

PSP Offset	Vector for Interrupt
0AH	Interrupt 22H (terminate routine)
0EH	Interrupt 23H (Control-C handler)
12H	Interrupt 24H (critical error handler) (versions 2.0 and later.)

- The minimum amount of memory a process can reserve is 6 paragraphs (60H bytes), which constitutes the initial portion of the process's PSP (including the reserved areas).
- The amount of memory required by the program is not necessarily the same as the size of the file that holds the program on disk. The program must allow for its PSP and stack in the amount of memory reserved; on the other hand,the memory occupied by code and data used only during program initialization frequently can be discarded as a side effect of the Function 31H call.

  Before Function 31H is called, memory allocated to the terminating process's environment block should be released by loading ES with the segment value at offset 2CH in the PSP (the segment address of the environment) and calling Function 49H (Free Memory Block).
- The terminating process should return a completion code in the AL register. If the program terminates normally, the return code should be 00H. A return code of 01H or greater usually indicates that termination was caused by an error encountered by the process.

The parent process can retrieve the return code with Function 4DH (Get Return Code of Child Process). If control returns to COMMAND.COM, the return code can be tested with an ERRORLEVEL statement in a batch file.

- After terminating the current process, MS-DOS attempts to set the program's memory allocation to the amount specified in DX.
- Function 31H is most often used for memory-resident utilities and subroutine libraries that can be accessed using interrupts.
- This function is preferable to Interrupt 27H (Terminate and Stay Resident) because it allows programs that are larger than 64 KB to remain resident, allows the terminating program to pass a return code to the parent process, and does not require that the CS register contain the PSP address.

Related Functions

48H (Allocate Memory Block)
49H (Free Memory Block)
4AH (Resize Memory Block)
4BH (Load and Execute Program)
4CH (Terminate Process with Return Code)
4DH (Get Return Code of Child Process)

Example

```
        ;************************************************************;
        ;                                                            ;
        ;          Function 31H: Terminate and Stay Resident         ;
        ;                                                            ;
        ;          void keep_process(exit_code,nparas)               ;
        ;               int exit_code,nparas;                        ;
        ;                                                            ;
        ;          Does NOT return!                                  ;
        ;                                                            ;
        ;************************************************************;

cProc   keep_process,PUBLIC
parmB   exit_code
parmW   nparas
cBegin
        mov     al,exit_code    ; Get return code.
        mov     dx,nparas       ; Set DX to number of paragraphs the
                                ; program wants to keep.
        mov     ah,31h          ; Set function code.
        int     21h             ; Ask MS-DOS to keep process.
cEnd
```

Interrupt 21H (33)
Function 33H (51)

2.0 and later

Get/Set Control-C Check Flag

Function 33H gets or sets the status of the Control-C check flag.

To Call

AH = 33H
AL = 00H get current Control-C check flag
     01H set Control-C check flag to value in DL

If AL is 01H:

DL = 00H set Control-C check flag to off
     01H set Control-C check flag to on

Returns

AL = 00H flag set successfully
     FFH code in AL on call not 00H or 01H

If AL was 00H on call:

DL = 00H Control-C check flag off
     01H Control-C check flag on

Programmer's Notes

- If the Control-C check flag is off, MS-DOS checks for a Control-C entered at the keyboard only during servicing of the character I/O functions, 01H through 0CH. If the Control-C check flag is on, MS-DOS also checks for user entry of a Control-C during servicing of other functions, such as file and record operations.
- The state of the Control-C check flag affects all programs. If a program needs to change the state of Control-C checking, it should save the original flag and restore it before terminating.

Related Functions

None

Example

```
        ;*************************************************************;
        ;                                                             ;
        ;          Function 33H: Get/Set Control-C Check Flag         ;
        ;                                                             ;
        ;          int controlC(func,state)                           ;
        ;              int func,state;                                ;
        ;                                                             ;
        ;          Returns current state of Control-C flag.           ;
        ;                                                             ;
        ;*************************************************************;

cProc   controlC,PUBLIC
parmB   func
parmB   state
cBegin
        mov     al,func         ; Get set/reset function.
        mov     dl,state        ; Get new value if present.
        mov     ah,33h          ; MS-DOS ^C check function.
        int     21h             ; Call MS-DOS.
        mov     al,dl           ; Return current state.
        cbw                     ; Clear high byte of return value.
cEnd
```

Interrupt 21H (33) Function 34H (52)

2.0 and later

Return Address of InDOS Flag

Function 34H returns the address of the InDOS flag, which reflects the current state of Interrupt 21H function processing.

Note: Microsoft cannot guarantee that the information in this entry will be valid for future versions of MS-DOS.

To Call

AH = 34H

Returns

ES:BX = segment:offset of InDOS flag

Programmer's Notes

- The InDOS flag is a byte within the MS-DOS kernel. The value in InDOS is incremented when MS-DOS begins execution of an Interrupt 21H function and decremented when MS-DOS's processing of that function is completed. Thus, the value of InDOS is zero only when no Interrupt 21H processing is occurring.
- The InDOS flag is one of the elements used in terminate-and-stay-resident (TSR) programs to determine when the TSR can be executed safely.

Related Functions

None

Example

```
        ;*********************************************************;
        ;                                                         ;
        ;       Function 34H: Get Return Address of InDOS Flag    ;
        ;                                                         ;
        ;       char far *inDOS_ptr()                             ;
        ;                                                         ;
        ;       Returns a far pointer to the MS-DOS inDOS flag.   ;
        ;                                                         ;
        ;*********************************************************;

cProc   inDOS_ptr,PUBLIC
cBegin
        mov     ah,34h          ; InDOS flag function.
        int     21h             ; Call MS-DOS.
        mov     ax,bx           ; Return offset in AX.
        mov     dx,es           ; Return segment in DX.
cEnd
```

Interrupt 21H (33) Function 35H (53)

2.0 and later

Get Interrupt Vector

Function 35H returns the address stored in the interrupt vector table for the handler associated with the specified interrupt.

To Call

AH = 35H
AL = interrupt number

Returns

ES:BX = segment:offset of handler for interrupt specified in AL

Programmer's Note

- Interrupt vectors should always be read with Function 35H and set with Function 25H (Set Interrupt Vector). Programs should never attempt to read or change interrupt vectors directly in memory.

Related Function

25H (Set Interrupt Vector)

Example

```
        ;************************************************************;
        ;                                                            ;
        ;           Function 35H: Get Interrupt Vector               ;
        ;                                                            ;
        ;           typedef void (far *FCP)();                       ;
        ;           FCP get_vector(intnum)                           ;
        ;               int intnum;                                  ;
        ;                                                            ;
        ;           Returns a far code pointer that is the           ;
        ;           segment:offset of the interrupt vector.          ;
        ;                                                            ;
        ;************************************************************;

cProc   get_vector,PUBLIC
parmB   intnum
cBegin
        mov     al,intnum       ; Get interrupt number into AL.
        mov     ah,35h          ; Select "get vector" function.
        int     21h             ; Call MS-DOS.
        mov     ax,bx           ; Return vector offset.
        mov     dx,es           ; Return vector segment.
cEnd
```

Interrupt 21H (33) Function 36H (54)

2.0 and later

Get Disk Free Space

Function 36H returns disk-storage information for the specified drive.

To Call

AH = 36H
DL = drive specification (0 = default drive, 1 = drive A, 2 = drive B, and so on)

Returns

If function is successful:

AX = number of sectors per cluster
BX = number of clusters available
CX = number of bytes per sector
DX = number of clusters on drive

If function is not successful:

AX = FFFFH invalid drive number in DL

Programmer's Notes

- The AX register should be checked for a value of FFFFH (error) before information returned by this function is used.
- The number of bytes of free storage remaining on the disk can be calculated by multiplying available clusters times sectors per cluster times bytes per sector (BX * AX * CX).
- Function 36H regards "lost" clusters (clusters that are allocated in the file allocation table [FAT] but do not belong to a file) as being in use and subtracts them from the amount of available storage, exactly as if they were allocated to a file.
- With MS-DOS versions 2.0 and later, Function 36H should be used in preference to the FCB Functions 1BH (Get Default Drive Data) and 1CH (Get Drive Data).

Related Functions

1BH (Get Default Drive Data)
1CH (Get Drive Data)

Example

```
        ;*********************************************************;
        ;                                                         ;
        ;             Function 36H: Get Disk Free Space           ;
        ;                                                         ;
        ;             long free_space(drive_ltr)                  ;
        ;                 char drive_ltr;                         ;
        ;                                                         ;
        ;             Returns the number of bytes free as         ;
        ;             a long integer.                             ;
        ;                                                         ;
        ;*********************************************************;

cProc   free_space,PUBLIC
parmB   drive_ltr
cBegin
        mov     dl,drive_ltr    ; Get drive letter.
        or      dl,dl           ; Leave 0 alone.
        jz      fsp
        and     dl,not 20h      ; Convert letter to uppercase.
        sub     dl,'A'-1        ; Convert to drive number: 'A' = 1,
                                ; 'B' = 2, etc.
fsp:
        mov     ah,36h          ; Set function code.
        int     21h             ; Ask MS-DOS to get disk information.
        mul     cx              ; Bytes/sector * sectors/cluster
        mul     bx              ; * free clusters.
cEnd
```

Interrupt 21H (33) Function 38H (56)

2.0 and later

Get/Set Current Country: Get Current Country

Function 38H includes two subfunctions that either get or set country data, depending on the value in the DX register when the function is called.

With MS-DOS versions 2.0 and later, if DX contains any value other than FFFFH, the Get Current Country subfunction is invoked. Information on date, currency, and other country-specific formats is then returned in a buffer specified by the calling program. The country code is usually the same as the country's international telephone prefix.

To Call

AH = 38H

With MS-DOS versions 2.x:

AL	= 00H	current country
DS:DX	= segment:offset of 32-byte buffer	

With MS-DOS versions 3.x:

AL	= 00H	current country
	01–FEH	country code between 1 and 254
	FFH	country code of 255 or greater, specified in BX
BX	= country code if AL = FFH	
DS:DX	= segment:offset of 34-byte buffer	

Returns

If function is successful:

Carry flag is clear.

BX	= country code (MS-DOS version 3.x only)
DS:DX	= segment:offset of buffer containing country information

If function is not successful:

Carry flag is set.

AX	= error code:	
	02H	invalid country code

Programmer's Notes

- With MS-DOS versions 2.x, the Get Current Country subfunction returns the following information for the current country in the 32-byte country-data buffer (ASCIIZ format is an ASCII character string ending in a zero byte):

Offset	Type	Description
00H	Word	Date format: 0 = United States (m/d/y) 1 = Europe (d/m/y) 2 = Japan (y/m/d)
02H	ASCIIZ	Currency symbol
04H	ASCIIZ	Character used as thousands separator
06H	ASCIIZ	Character used as decimal separator
08H	24 bytes	Reserved

- With MS-DOS versions 3.x, the Get Current Country subfunction returns the following information for the specified country in the 34-byte country-data buffer:

Offset	Type	Description
00H	Word	Date format: 0 = United States (m/d/y) 1 = Europe (d/m/y) 2 = Japan (y/m/d)
02H	ASCIIZ	Currency symbol (5 bytes, as opposed to 2 in versions 2.x of MS-DOS)
07H	ASCIIZ	Character used as thousands separator
09H	ASCIIZ	Character used as decimal separator
0BH	ASCIIZ	Character used as date separator
0DH	ASCIIZ	Character used as time separator
0FH	Byte	Position of currency symbol; possible values are 00H Currency symbol precedes value with no space 01H Currency symbol follows value with no space 02H Currency symbol precedes value with one space 03H Currency symbol follows value with one space
10H	Byte	Number of decimal places in currency

(more)

Offset	Type	Description
11H	Byte	Time format (00H = 12-hour clock; 01H = 24-hour clock)
12H	Dword	Case-mapping call address (*See* Programmer's Notes below.)
16H	ASCIIZ	Character used as separator in data lists
18H	10 bytes	Reserved

- The case-mapping call address (MS-DOS versions 3.x only) is the segment:offset of a FAR procedure that performs country-specific mapping on ASCII characters in the range 80H through 0FFH. The character to be mapped must be placed in the AL register before the call is made. If the character has an uppercase value, that value is returned in AL. If the character has no such value, AL is unchanged.
- Function 59H (Get Extended Error Information) provides further information on any error—in particular, the code, class, recommended corrective action, and locus of the error.

Related Function

38H (Set Current Country subfunction)

Example

```
        ;************************************************************;
        ;                                                            ;
        ;         Function 38H: Get/Set Current Country Data         ;
        ;                                                            ;
        ;         int country_info(country,pbuffer)                  ;
        ;             char country,*pbuffer;                         ;
        ;                                                            ;
        ;         Returns -1 if the "country" code is invalid.       ;
        ;                                                            ;
        ;************************************************************;

cProc   country_info,PUBLIC,ds
parmB   country
parmDP  pbuffer
cBegin
        mov     al,country      ; Get country code.
        loadDP  ds,dx,pbuffer   ; Get buffer pointer (or -1).
        mov     ah,38h          ; Set function code.
        int     21h             ; Ask MS-DOS to get country
                                ; information.
        jnb     cc_ok           ; Branch if country code OK.
        mov     ax,-1           ; Else return -1.
cc_ok:
cEnd
```

Interrupt 21H (33) Function 38H (56)

3.0 and later

Get/Set Current Country: Set Current Country

Function 38H includes two subfunctions that either get or set country data, depending on the value in the DX register when the function is called.

With MS-DOS versions 3.0 and later, the Set Current Country subfunction is invoked if Function 38H is called with DX = FFFFH (–1). This subfunction selects the country for which subsequent calls to Get Current Country will return information. The country code used with this function is usually the same as the country's international telephone prefix.

To Call

AH = 38H
AL = country code for a code less than 255
FFH for country code of 255 or greater, specified in BX
BX = country code if AL = FFH
DX = FFFFH (–1)

Returns

If function is successful:

Carry flag is clear.

If function is not successful:

Carry flag is set.

AX = error code:
02H invalid country code

Programmer's Notes

- MS-DOS normally uses the country code associated with the current KEYBxx keyboard driver file, if any. Otherwise, the default country code is OEM dependent.
- Function 59H (Get Extended Error Information) provides further information on any error—in particular, the code, class, recommended corrective action, and locus of the error.

Related Function

38H (Get Current Country subfunction)

Example

See Function 38H Subfunction Get Current Country for example.

Interrupt 21H (33) Function 39H (57)

2.0 and later

Create Directory

Function 39H creates a subdirectory using the specified path.

To Call

AH = 39H
DS:DX = segment:offset of ASCIIZ path

Returns

If function is successful:

Carry flag is clear.

If function is not successful:

Carry flag is set.

AX = error code:
- 03H path not found
- 05H access denied

Programmer's Notes

- The path must be a null-terminated ASCII string (ASCIIZ).
- MS-DOS places the current directory (.) and parent directory (..) entries in all new directories.
- Function 39H returns error code 05H (access denied) in the following cases:
  - File or directory with the same name already exists in the specified path.
  - Parent directory is the root directory and the root directory is full.
  - Path specifies a device.
  - Program is running on a network under MS-DOS version 3.1 or later and the user does not have Create access to the parent directory.
- Function 59H (Get Extended Error Information) provides further information on any error—in particular, the code, class, recommended corrective action, and locus of the error.

Related Functions

3AH (Remove Directory)
3BH (Change Current Directory)
47H (Get Current Directory)

Example

```
        ;*************************************************************;
        ;                                                             ;
        ;              Function 39H: Create Directory                 ;
        ;                                                             ;
        ;              int make_dir(pdirpath)                         ;
        ;                  char *pdirpath;                            ;
        ;                                                             ;
        ;              Returns 0 if directory created OK,             ;
        ;              otherwise returns error code.                  ;
        ;                                                             ;
        ;*************************************************************;

cProc   make_dir,PUBLIC,ds
parmDP  pdirpath
cBegin
        loadDP  ds,dx,pdirpath  ; Get pointer to pathname.
        mov     ah,39h          ; Set function code.
        int     21h             ; Ask MS-DOS to make new subdirectory.
        jb      md_err          ; Branch on error.
        xor     ax,ax           ; Else return 0.
md_err:
cEnd
```

Interrupt 21H (33) Function 3AH (58)

2.0 and later

Remove Directory

Function 3AH removes (deletes) the specified subdirectory.

To Call

AH = 3AH
DS:DX = segment:offset of ASCIIZ path

Returns

If function is successful:

Carry flag is clear.

If function is not successful:

Carry flag is set.

AX = error code:

03H	path not found
05H	access denied
10H	current directory was specified

Programmer's Notes

- The path must be a null-terminated ASCII string (ASCIIZ).
- Function 3AH returns error code 05H (access denied) in the following cases:
  - Directory is not empty.
  - Root directory was specified.
  - Current directory was specified.
  - Path does not specify a valid directory.
  - Directory is malformed (. and .. not first two entries).
  - User has insufficient access rights on a network running under MS-DOS version 3.1 or later.
- Function 59H (Get Extended Error Information) provides further information on any error — in particular, the code, class, recommended corrective action, and locus of the error.

Related Functions

39H (Create Directory)
3BH (Change Current Directory)
47H (Get Current Directory)

Example

```
        ;************************************************************;
        ;                                                            ;
        ;               Function 3AH: Remove Directory               ;
        ;                                                            ;
        ;               int remove_dir(pdirpath)                     ;
        ;                   char *pdirpath;                          ;
        ;                                                            ;
        ;               Returns 0 if directory was removed,          ;
        ;               otherwise returns error code.                ;
        ;                                                            ;
        ;************************************************************;

cProc   remove_dir,PUBLIC,ds
parmDP  pdirpath
cBegin
        loadDP  ds,dx,pdirpath  ; Get pointer to pathname.
        mov     ah,3ah          ; Set function code.
        int     21h             ; Ask MS-DOS to delete subdirectory.
        jb      rd_err          ; Branch on error.
        xor     ax,ax           ; Else return 0.
rd_err:
cEnd
```

Interrupt 21H (33) Function 3BH (59)

2.0 and later

Change Current Directory

Function 3BH changes the current directory to the specified path.

To Call

AH = 3BH
DS:DX = segment:offset of ASCIIZ path

Returns

If function is successful:

Carry flag is clear.

If function is not successful:

Carry flag is set.

AX = error code:
03H path not found

Programmer's Notes

- The path must be a null-terminated ASCII string (ASCIIZ).
- Before a call to Function 3BH, Function 47H (Get Current Directory) can be used to determine the current directory so that the original directory can be restored later (for example, on termination of the program).
- Function 3BH can be used with programs that rely on either FCB-based or handle-based calls. It is the only method of changing the current directory that is supported by MS-DOS.
- The path string is limited to a total of 64 characters, including separators.
- Function 59H (Get Extended Error Information) provides further information on any error — in particular, the code, class, recommended corrective action, and locus of the error.

Related Functions

39H (Create Directory)
3AH (Remove Directory)
47H (Get Current Directory)

Example

```
;*************************************************************;
;                                                             ;
;            Function 3BH: Change Current Directory           ;
;                                                             ;
;            int change_dir(pdirpath)                         ;
;                char *pdirpath;                              ;
;                                                             ;
;            Returns 0 if directory was changed,              ;
;            otherwise returns error code.                    ;
;                                                             ;
;*************************************************************;

cProc   change_dir,PUBLIC,ds
parmDP  pdirpath
cBegin
        loadDP  ds,dx,pdirpath  ; Get pointer to pathname.
        mov     ah,3bh          ; Ask MS-DOS to move to
        int     21h             ; different directory.
        jb      cd_err          ; Branch on error.
        xor     ax,ax           ; Else return 0.
cd_err:
cEnd
```

Interrupt 21H (33) Function 3CH (60)

2.0 and later

Create File with Handle

Function 3CH creates a file, assigns it the attributes specified, and returns a 16-bit handle for the file. If the named file already exists, Function 3CH opens it and truncates it to zero length.

To Call

AH	= 3CH
CX	= attribute
DS:DX	= segment:offset of ASCIIZ pathname

Returns

If function is successful:

Carry flag is clear.

AX = handle number

If function is not successful:

Carry flag is set.

AX = error code:

03H	path not found
04H	too many open files
05H	access denied

Programmer's Notes

- Function 3CH is preferable to Function 16H (Create File with FCB) for creating a file because it supports full pathnames. Function 16H should be used only if compatibility with versions 1.x of MS-DOS is required.
- The pathname must be a null-terminated ASCII string (ASCIIZ).
- Bits 0 through 2 of the 2-byte file attribute in CX determine whether the file is normal, read-only, hidden, or system. The attribute codes are
  - 00H normal file
  - 01H read-only file
  - 02H hidden file
  - 04H system file

  Bits 3 through 5 are associated with volume labels, subdirectories, and archive files. The volume and subdirectory bits are invalid for Function 3CH and must be set to 0. Bits 6 through 15 should be set to 0 to ensure future compatibility.

Values can be combined to set several file attributes. For example, if Function 3CH is called with CX = 0003H, the file created is a read-only hidden file.

- Because Function 3CH truncates an existing file to zero length, any information previously in the file is lost. Alternative functions that protect against such loss include the following:
  - Function 3DH (Open File with Handle) or Function 4EH (Find First File), which can be used to check for the previous existence of the file before Function 3CH is called
  - Function 5AH (Create Temporary File), which creates a file in the specified subdirectory and gives it a unique name assigned by MS-DOS
  - Function 5BH (Create New File), which is similar to Function 3CH but fails if it finds a file that matches the specified pathname
- After creating a file, Function 3CH sets the position of the file pointer to 0. Thus, the next read or write operation takes place at the beginning of the file.
- Function 3CH returns error code 04H (too many open files) if no handle is currently available. With MS-DOS versions 3.2 and earlier, a single process can have no more than 20 files open at one time, 5 of which are normally assigned to the standard devices.

  Error code 05H (access denied) is returned if the file is to be created in the root directory and the root is full or if a read-only file with the same name already exists in the specified subdirectory.
- On networks running under MS-DOS version 3.1 or later, the user must have Create access to the directory containing the file specified.
- Function 59H (Get Extended Error Information) provides further information on any error — in particular, the code, class, recommended corrective action, and locus of the error.

Related Functions

16H (Create File with FCB)
43H (Get/Set File Attributes)
5AH (Create Temporary File)
5BH (Create New File)

Example

```
;*********************************************************;
;                                                         ;
;           Function 3CH: Create File with Handle         ;
;                                                         ;
;           int create(pfilepath,attr)                    ;
;              char *pfilepath;                           ;
;           int attr;                                     ;
;                                                         ;
;           Returns -1 if file was not created,           ;
;           otherwise returns file handle.                ;
;                                                         ;
;*********************************************************;
```

(more)

```
cProc   create,PUBLIC,ds
parmDP  pfilepath
parmW   attr
cBegin
        loadDP  ds;dx,pfilepath ; Get pointer to pathname.
        mov     cx,attr         ; Get new file's attribute.
        mov     ah,3ch          ; Ask MS-DOS to make a new file.
        int     21h
        jnb     cr_ok           ; Branch if MS-DOS returned handle.
        mov     ax,-1           ; Else return -1.
cr_ok:
cEnd
```

Interrupt 21H (33) Function 3DH (61)

2.0 and later

Open File with Handle

Function 3DH opens the specified file and returns a 16-bit handle number for subsequent access to the file.

To Call

AH = 3DH

With versions 2.x of MS-DOS:

AL = file-access code:

Bits	Value	Meaning
3–7	00000	Reserved
0–2	000	Read-only access
	001	Write-only access
	010	Read/write access

DS:DX = segment:offset of ASCIIZ pathname

With versions 3.x of MS-DOS:

AL = file-access, file-sharing, and inheritance codes:

Bits	Value	Meaning
7 (inherit bit)	0	Child process inherits file
	1	Child process does not inherit file
4–6 (sharing mode; file access granted to other processes)	000	Compatibility mode
	001	Deny read/write access
	010	Deny write access
	011	Deny read access
	100	Deny none
3	0	Reserved
0–2 (access code; file usage)	000	Read-only access
	001	Write-only access
	010	Read/write access

DS:DX = segment:offset of ASCIIZ pathname

Returns

If function is successful:

Carry flag is clear.

AX = handle number

If function is not successful:

Carry flag is set.

AX = error code:

02H	file not found
03H	path not found
04H	too many open files
05H	access denied
0CH	invalid access code

Programmer's Notes

- Function 3DH is preferable to Function 0FH (Open File with FCB) because it allows the use of pathnames. Function 0FH should be used only if compatibility with versions 1.x of MS-DOS is required.
- Function 3DH opens any file matching the pathname in DS:DX, including hidden and system files.
- The pathname must be a null-terminated ASCII string (ASCIIZ).
- Function 3DH returns error code 04H (too many open files) if no handle is currently available. With MS-DOS versions 3.2 and earlier, a single process can have no more than 20 files open at one time, 5 of which are normally assigned to the standard devices.

  Function 3DH returns error code 05H (access denied) if the pathname specifies a directory or volume label or if read/write access was requested for a read-only file.

  Function 3DH returns error code 0CH (invalid access code) if bits 0–2 in AL contain any value other than 000, 001, or 010.
- With MS-DOS versions 2.x, only bits 0–2 of the byte in AL are meaningful; they should contain the type of access allowed for the file. Bits 3–7 should always be zero.

  With MS-DOS versions 3.0 and later, networking capabilities require bits 4–7, as well as 0–2, to be set. (Bit 3 is reserved and should be 0.)

  Bit 7, the inherit bit, should be set to indicate whether child processes created by the current process with Function 4BH (Load and Execute Program) either can (0) or cannot (1) inherit the file. When a process inherits a file, it also inherits the access and sharing modes.

Bits 4–6 are called the "sharing code"; they indicate the type of access other users on the network can have to the file. The five sharing modes and the conditions under which they pertain are as follows:

- mode 000 (compatibility). Allows other programs running on the same machine unlimited access to the file. Programs running on other machines cannot access the file across the network unless it has the read-only attribute. An attempt to open the file in compatibility mode fails if the file has already been opened with any other sharing mode.
- 001 (deny read and write access). Provides exclusive access to the file. Any subsequent attempts by others (including the current process) to open the file fail. This mode fails if the file has already been opened in compatibility mode or for read or write access, even by the current process.
- 010 (deny write access). Allows other processes to open the file for read-only access. This mode fails if the file has already been opened in compatibility mode or for write access by any other process.
- 011 (deny read access). Allows other processes to open the file for write-only access. This mode fails if the file has already been opened in compatibility mode or for read access by any other process.
- 100 (deny none). Similar to compatibility mode, but does not allow other processes to open the file in compatibility mode. This mode fails if the file has already been opened in compatibility mode by any other process.

- When the file is opened, the position of the file pointer is set to 0. Function 42H (Move File Pointer) can be used to change its position.
- With MS-DOS versions 3.0 and later, if this function fails because of a file-sharing error, the operating system issues an Interrupt 24H (Critical Error Handler Address) with error code 02H (drive not ready). Function 59H (Get Extended Error Information) must be used to find the extended error code specifying the type of sharing violation that occurred.

Related Functions

0FH (Open File with FCB)
3EH (Close File)
3FH (Read File or Device)
40H (Write File or Device)
42H (Move File Pointer)
43H (Get/Set File Attributes)
57H (Get/Set Date/Time of File)

Example

```
;*************************************************************;
;                                                             ;
;            Function 3DH: Open File with Handle              ;
;                                                             ;
;            int open(pfilepath,mode)                         ;
;                char *pfilepath; int mode;                   ;
;                                                             ;
;            Modes:                                           ;
;                    0: Read                                  ;
;                    1: Write                                 ;
;                    2: Read/Write                            ;
;                                                             ;
;            Returns -1 if file was not opened,               ;
;            otherwise returns file handle.                   ;
;                                                             ;
;*************************************************************;

cProc   open,PUBLIC,ds
parmDP  pfilepath
parmB   mode
cBegin
        loadDP  ds,dx,pfilepath ; Get pointer to pathname.
        mov     al,mode         ; Get read/write mode.
        mov     ah,3dh          ; Request MS-DOS to open the
        int     21h             ; existing file.
        jnb     op_ok           ; Branch if MS-DOS returned handle.
        mov     ax,-1           ; Else return -1.
op_ok:
cEnd
```

Interrupt 21H (33) Function 3EH (62)

2.0 and later

Close File

Function 3EH closes the file referenced by the specified handle.

To Call

AH = 3EH
BX = handle number

Returns

If function is successful:

Carry flag is clear.

If function is not successful:

Carry flag is set.

AX = error code:
06H invalid handle number

Programmer's Notes

- The handle in BX must be one that was returned by a successful call to one of the following functions:
  - 3CH (Create File with Handle)
  - 3DH (Open File with Handle)
  - 5AH (Create Temporary File)
  - 5BH (Create New File)
- If the file has been modified, truncated, or extended, Function 3EH updates the current date, time, and file size in the directory entry.
- All internal MS-DOS buffers for the file, including directory and file allocation table (FAT) buffers, are flushed to disk.
- With MS-DOS versions 3.0 and later, a program must remove all file locks in effect before it closes a file. The result of closing a file with active locks is unpredictable.
- Function 59H (Get Extended Error Information) provides further information on any error — in particular, the code, class, recommended corrective action, and locus of the error.

Related Functions

10H (Close File with FCB)
3CH (Create File with Handle)
3DH (Open File with Handle)
5AH (Create Temporary File)
5BH (Create New File)

Example

```
        ;*********************************************************;
        ;                                                         ;
        ;               Function 3EH: Close File                  ;
        ;                                                         ;
        ;               int close(handle)                         ;
        ;                   int handle;                           ;
        ;                                                         ;
        ;               Returns -1 if file was not closed,        ;
        ;               otherwise returns 0.                      ;
        ;                                                         ;
        ;*********************************************************;

cProc   close,PUBLIC
parmW   handle
cBegin
        mov     bx,handle       ; Get handle.
        mov     ah,3eh          ; Set function codes.
        int     21h             ; Ask MS-DOS to close handle.
        mov     al,0
        jnb     cl_ok           ; Branch if no error.
        mov     al,-1           ; Else return -1.
cl_ok:
        cbw                     ; Extend result.
cEnd
```