Using Debug

1 INTRODUCING DEBUG

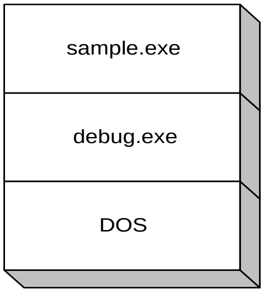
As you begin to learn assembly language programming, the importance of using a program called a *debugger* cannot be stressed too much. A debugger displays the contents of memory and lets you view registers and variables as they change. You can step through a program one line at a time (called *tracing*), making it easier to find logic errors. In this appendix, we offer a tutorial on using the debug.exe program that is supplied with both DOS and Windows (located in the \Windows\Command directory). From now on, we will just call this program *Debug*. Later, you will probably want to switch to a more sophisticated debugger such as Microsoft CodeView or Borland Turbo Debugger. But for now, Debug is the perfect tool for writing short programs and getting acquainted with the Intel microprocessor.

You can use Debug to test assembler instructions, try out new programming ideas, or to carefully step through your programs. It takes supreme overconfidence to write an assembly language program and run it directly from DOS the first time! If you forget to match pushes and pops, for example, a return from a subroutine will branch to an unexpected location. Any call or jump to a location outside your program will almost surely cause the program to crash. For this reason, you would be wise to run any new program you've written in Debug. Trace the program one line at a time, and watch the stack pointer (SP) very closely as you step through the program, and note any unusual changes to the CS and IP registers. Particularly when CS takes on a new value, you should be suspicious that your program has branched into the *Twilight Zone*®.

Debugging functions. Some of the most rudimentary functions that any debugger can perform are the following:

- Assemble short programs
- View a program's source code along with its machine code
- View the CPU registers and flags
- Trace or execute a program, watching variables for changes
- Enter new values into memory
- Search for binary or ASCII values in memory
- Move a block of memory from one location to another
- Fill a block of memory
- Load and write disk files and sectors

Many commercial debuggers are available for Intel microprocessors, ranging widely in sophistication and cost: CodeView, Periscope, Atron, Turbo Debugger, SYMDEB, Codesmith-86, and Advanced-Trace-86, to mention just a few. Of these, Debug is the simplest. The basic principles learned using Debug may then be applied to nearly any other debugger.



Debug is called an *assembly level* debugger because it displays only assembly mnemonics and machine instructions. Even if you use it to debug a compiled C++ program, for example, you will not see the program's source code. Instead, you will see a disassembly of the program's machine instructions.

To trace or execute a machine language program with Debug, type the name of the program as a command line parameter. For example, to debug the program sample.exe, you would type the following command line at the DOS prompt:

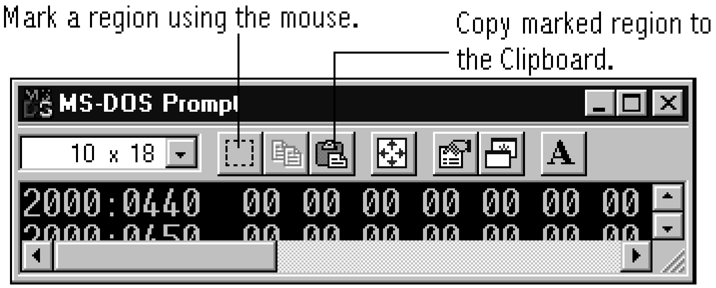
```
debug sample.exe
```

If we could picture DOS memory after typing this command, we would see DOS loaded in the lowest area, debug.exe loaded above DOS, and the program sample.exe loaded above Debug. In this way, several programs are resident in memory at the same time. DOS retains control over the execution of Debug, and Debug controls the execution of sample.exe.

Printing a Debugging Session (local printer only). If you have a printer attached directly to your computer, you can get a printed copy of everything you're doing during a debugging session by pressing the Ctrl-PrtScrn keys. This command is a toggle, so it can be typed a second time to turn the printer output off.

Printing a Debugging Session (network printer). If your computer is attached to a network and there is a printer on the network, printing a debugging session is a bit challenging. The best way we've found is to prepare a script file containing all the debug commands you plan to type. Run Debug, telling it to read its input from the script file, and have Debug send the output to another file. Then, print the output file in the same way you usually print on the network. In Windows, for example, the output file can be loaded into *Notepad* and printed from there. See the section later in this appendix entitled *Using Script Files with Debug.*

Using the Mark and Copy Operations in a DOS Window. Under Windows, when you run Debug in a window, a toolbar has commands that you can use to mark a section of the window, copy it to the clipboard, and paste it into some other application (such as Notepad or Word):


Mark a region using the mouse.
Copy marked region to the Clipboard.
MS-DOS Prompt
10 x 18
A

```
2000:0440   00 00 00 00 00 00 00
```

2 DEBUG COMMAND SUMMARY

Debug commands may be divided into four categories: program creation/debugging, memory manipulation, miscellaneous, and input-output:

Program Creation and Debugging

A	Assemble a program using instruction mnemonics
G	Execute the program currently in memory
R	Display the contents of registers and flags
P	Proceed past an instruction, procedure, or loop
T	Trace a single instruction
U	Disassemble memory into assembler mnemonics

Memory Manipulation

C	Compare one memory range with another
D	Dump (display) the contents of memory
E	Enter bytes into memory
F	Fill a memory range with a single value
M	Move bytes from one memory range to another
S	Search a memory range for specific value(s)

Miscellaneous

H	Perform hexadecimal addition and subtraction
Q	Quit Debug and return to DOS

Input-Output

I	Input a byte from a port
L	Load data from disk
O	Send a byte to a port
N	Create a filename for use by the L and W commands
W	Write data from memory to disk

Default Values. When Debug is first loaded, the following defaults are in effect:

1. All segment registers are set to the bottom of free memory, just above the debug.exe program.

2. IP is set to 0100h.

3. Debug reserves 256 bytes of stack space at the end of the current segment.

4. All of available memory is allocated (reserved).

5. BX:CX are set to the length of the current program or file.

6. The flags are set to the following values: NV (Overflow flag clear), UP (Direction flag = up), EI (interrupts enabled), PL (Sign flag = positive), NZ (Zero flag clear), NA (Auxiliary Carry flag clear), PO (odd parity), NC (Carry flag clear).

2.1 Command Parameters

Debug's command prompt is a hyphen (–). Commands may be typed in either uppercase or lowercase letters, in any column. A command may be followed by one or more parameters. A comma or space may be used to separate any two parameters. The standard command parameters are explained here.

Address. A complete segment-offset address may be given, or just an offset. The segment portion may be a hexadecimal number or register name. For example:

F000:100	**Segment, offset**
DS:200	**Segment register, offset**
0AF5	**Offset**

Filespec. A file specification, made up of a drive designation, filename, and extension. At a minimum, a filename must be supplied. Examples are:

```
b:prog1.com
c:\asm\progs\test.com
file1
```

List. One or more byte or string values:

```
10,20,30,40
'A','B',50
```

Range. A *range* refers to a span of memory, identified by addresses in one of two formats. In Format 1, if the second address is omitted, it defaults to a standard value. In Format 2, the value following the letter L is the number of bytes to be processed by the command. A range cannot be greater than 10000h (65,536):

Syntax	Examples
Format 1: address [,address]	**100,500 CS:200,300 200**

`Format 2: address L [value]  100 L 20` ***(Refers to the 20h bytes starting at location 100.)***

Sector. A sector consists of a starting sector number and the number of sectors to be loaded or written. You can access logical disk sectors Using the L (load) and W (write) commands.

String. A string is a sequence of characters enclosed in single or double quotes. For example:

```
'COMMAND'
"File cannot be opened."
```

Value. A value consists of a 1- to 4-character hexadecimal number. For example:

```
3A
3A6F
```

3 INDIVIDUAL COMMANDS

This section describes the most common Debug commands. A good way to learn them is to sit at the computer while reading this tutorial and experiment with each command.

3.1 ? (Help)

Press ? at the Debug prompt to see a list of all commands. For example:

Figure 1. Debug's List of Commands.

```
assemble         A [address]
compare          C range address
dump             D [range]
enter            E address [list]
fill             F range list
go               G [=address] [addresses]
hex              H value1 value2
input            I port
load             L [address] [drive] [firstsector] [number]
move             M range address
name             N [pathname] [arglist]
output                 O port byte
proceed          P [=address] [number]
quit             Q
register         R [register]
```

```
search                 S range list
trace           T [=address] [value]
unassemble      U [range]
write           W [address] [drive] [firstsector] [number]
```

3.2 A (Assemble)

Assemble a program into machine language. Command formats:

```
A
A address
```

If only the offset portion of *address* is supplied, it is assumed to be an offset from CS. Here are examples:

Example	Description
`A 100`	Assemble at CS:100h.
`A`	Assemble from the current location.
`A DS:2000`	Assemble at DS:2000h.

When you press Enter at the end of each line, Debug prompts you for the next line of input. Each input line starts with a segment-offset address. To terminate input, press the Enter key on a blank line. For example:

```
-a 100
5514:0100 mov ah,2
5514:0102 mov dl,41
5514:0104 int 21
5514:0106
```

(bold text is typed by the programmer)

3.3 C (Compare)

The C command compares bytes between a specified range with the same number of bytes at a target address. Command format:

```
C range address
```

For example, the bytes between DS:0100 and DS:0105 are compared to the bytes at DS:0200:

```
C 100 105 200
```

The following is displayed by Debug:

```
1F6E:0100  74  00  1F6E:0200
1F6E:0101  15  C3  1F6E:0201
1F6E:0102  F6  0E  1F6E:0202
1F6E:0103  C7  1F  1F6E:0203
1F6E:0104  20  E8  1F6E:0204
1F6E:0105  75  D2  1F6E:0205
```

3.4 D (Dump)

The D command displays memory on the screen as single bytes in both hexadecimal and ASCII. Command formats:

```
D
D address
D range
```

If no address or range is given, the location begins where the last D command left off, or at location DS:0 if the command is being typed for the first time. If *address* is specified, it consists of either a segment-offset address or just a 16-bit offset. *Range* consists of the beginning and ending addresses to dump.

Example	**Description**
`D F000:0`	Segment-offset
`D ES:100`	Segment register-offset
`D 100`	Sffset

The default segment is DS, so the segment value may be left out unless you want to dump an offset from another segment location. A range may be given, telling Debug to dump all bytes within the range:

```
D 150 15A           (Dump DS:0150 through 015A)
```

Other segment registers or absolute addresses may be used, as the following examples show:

Example	Description
`D`	Dump 128 bytes from the last referenced location.
`D SS:0 5`	Dump the bytes at offsets 0-5 from SS.
`D 915:0`	Dump 128 bytes at offset zero from segment 0915h.
`D 0 200`	Dump offsets 0-200 from DS.
`D 100 L 20`	Dump 20h bytes, starting at offset 100h from DS.

Memory Dump Example. The following figure shows an example of a memory dump. The numbers at the left are the segment and offset address of the first byte in each line. The next 16 pairs of digits are the hexadecimal contents of each byte. The characters to the right are the ASCII representation of each byte. This dump appears to be machine language instructions, rather than displayable characters.

Dump of offsets 0100h through 017Fh in COMMAND.COM:

```
-D 100
1CC0:0100  83 7E A4 01 72 64 C7 46-F8 01 00 8B 76 F8 80 7A  .~$.rdGFx...vx.z
1CC0:0110  A5 20 73 49 80 7A A5 0E-75 06 C6 42 A5 0A EB 3D  % sI.z%.u.FB%.k=
1CC0:0120  8B 76 F8 80 7A A5 08 74-0C 80 7A A5 07 74 06 80  .vx.z%.t..z%.t..
1CC0:0130  7A A5 0F 75 28 FF 46 FA-8B 76 FA 8B 84 06 F6 8B  z%.u(.Fz.vz...v.
1CC0:0140  7E F8 3A 43 A5 75 0C 03-36 A8 F4 8B 44 FF 88 43  ~x:C%u..6(t.D..C
1CC0:0150  A5 EB 0A A1 06 F6 32 E4-3B 46 FA 77 D8 8B 46 F8  %k.!.v2d;FzwX.Fx
1CC0:0160  40 89 46 F8 48 3B 46 A4-75 A1 A1 06 F6 32 E4 3B  @.FxH;F$u!!.v2d;
1CC0:0170  46 FC B9 00 00 75 01 41-A1 A8 F4 03 46 FC 8B 16  F|9..u.A!(t.F|..
```

The following dump shows a different part of COMMAND.COM. Because memory at this point contains a list of command names, the ASCII dump is more interesting:

```
-D 3AC0
1CD6:3AC0  05 45 58 49 53 54 EA 15-00 04 44 49 52 01 FA 09
.EXISTj...DIR.z.
1CD6:3AD0  07 52 45 4E 41 4D 45 01-B2 0C 04 52 45 4E 01 B2   .RENAME.2..REN.2
1CD6:3AE0  0C 06 45 52 41 53 45 01-3D 0C 04 44 45 4C 01 3D   ..ERASE.=..DEL.=
1CD6:3AF0  0C 05 54 59 50 45 01 EF-0C 04 52 45 4D 00 04 01   ..TYPE.o..REM...
1CD6:3B00  05 43 4F 50 59 01 CC 1A-06 50 41 55 53 45 00 1F   .COPY.L..PAUSE..
1CD6:3B10  13 05 44 41 54 45 00 38-18 05 54 49 4D 45 00 CE   ..DATE.8..TIME.N
1CD6:3B20  18 04 56 45 52 00 57 0E-04 56 4F 4C 01 C8 0D 03   ..VER.W..VOL.H..
1CD6:3B30  43 44 01 A6 12 06 43 48-44 49 52 01 A6 12 03 4D   CD.&..CHDIR.&..M
1CD6:3B40  44 01 D9 12 06 4D 4B 44-49 52 01 D9 12 03 52 44   D.Y..MKDIR.Y..RD
1CD6:3B50  01 0E 13 06 52 4D 44 49-52 01 0E 13 06 42 52 45   ....RMDIR....BRE
1CD6:3B60  41 4B 00 92 17 07 56 45-52 49 46 59 00 C7 17 04   AK....VERIFY.G..
1CD6:3B70  53 45 54 00 0F 10 07 50-52 4F 4D 50 54 00 FA 0F   SET....PROMPT.z.
1CD6:3B80  05 50 41 54 48 00 A0 0F-05 45 58 49 54 00 C9 11   .PATH. ..EXIT.I.
1CD6:3B90  05 43 54 54 59 01 F7 11-05 45 43 48 4F 00 59 17   .CTTY.w..ECHO.Y.
1CD6:3BA0  05 47 4F 54 4F 00 96 16-06 53 48 49 46 54 00 56   .GOTO....SHIFT.V
```

```
1CD6:3BB0  16 03 49 46 00 50 15 04-46 4F 52 00 68 14 04 43   ..IF.P..FOR.h..C
1CD6:3BC0  4C 53 00 53 12 00 00 00-00 00 00 00 00 00 00 00   LS.S............
```

3.5 E (Enter)

The E command places individual bytes in memory. You must supply a starting memory location where the values will be stored. If only an offset value is entered, the offset is assumed to be from DS. Otherwise, a 32-bit address may be entered or another segment register may be used. Command formats are:

`E` *`address`* Enter new byte value at *address*.

`E` *`address list`* Replace the contents of one or more bytes starting at the specified *address*, with the values contained in the *list*.

To begin entering hexadecimal or character data at DS:100, type:

```
E 100
```

Press the space bar to advance to the next byte, and press the Enter key to stop. To enter a string into memory starting at location CS:100, type:

```
E CS:100 "This is a string."
```

3.6 F (Fill)

The F command fills a range of memory with a single value or list of values. The range must be specified as two offset addresses or segment-offset addresses. Command format:

`F` ***`range list`***

Here are some examples. The commas are optional:

Example	**Description**
`F 100 500,' '`	Fill locations 100 through 500 with spaces.
`F CS:300 CS:1000,FF`	Fill locations CS:300 through 1000 with hex FFh.
`F 100 L 20 'A'`	Fill 20h bytes with the letter 'A', starting at location 100.

3.7 G (Go)

Execute the program in memory. You can also specify a breakpoint, causing the program to stop at a given address. Command formats:

`G`
`G` ***`breakpoint`***

```
G = startAddr breakpoint
G = startAddr breakpoint1 breakpoint2 ...
```

Breakpoint is a 16- or 32-bit address at which the processor should stop, and *startAddr* is an optional starting address for the processor. If no breakpoints are specified, the program runs until it stops by itself and returns to Debug. Up to 10 breakpoints may be specified on the same command line. Examples:

Example	**Description**
`G`	Execute from the current location to the end of the program.
`G 50`	Execute from the current location and stop before the instruction at offset CS:50.
`G=10 50`	Begin execution at CS:10 and stop before the instruction at offset CS:50.

3.8 H (Hexarithmetic)

The H command performs addition and subtraction on two hexadecimal numbers. The command format is:

```
H value1 value2
```

For example, the hexadecimal values 1A and 10 are added and subtracted:

```
H 1A 10
2A 0A                (displayed by Debug)
```

3.9 I (Input)

The I command inputs a byte from a specified input/output port and displays the value in hexadecimal. The command format is:

```
I port
```

Where *port* is a port number between 0 and FFFF. For example, we input a byte from port 3F8 (one of the COM1 ports), and Debug returns a value of 00:

```
-I 3F8
00
```

3.10 L (Load)

The L command loads a file (or logical disk sectors) into memory at a given address. To read a file, you must first initialize its name with the N (Name) command. If *address* is

Table 1. Examples of the Load Instruction.

Example	Description
L	Load named file into memory at CS:0100
L DS:0200	Load named file into memory at DS:0200
L 100 2 A 5	Load five sectors from drive C, starting at logical sector number 0Ah.
L 100 0 0 2	Load two sectors into memory at CS:100, from the disk in drive A, starting at logical sector number 0.

omitted, the file is loaded at CS:100. Debug sets BX and CX to the number of bytes read. Command format:

```
L
L address
L address drive firstsector number
```

The first format, with no parameters, implies that you want to read from a file into memory at CS:0100. (Use the N command to name the file.) The second format also reads from a named file, but lets you specify the target address. The third format loads sectors from a disk drive, where you specify the drive number (0 = A, 1 = B, etc.), the first logical sector number, and the number of sectors to read. Examples are shown in Table 1.

Each sector is 512 bytes, so a sector loaded at offset 100 would fill memory through offset 2FF. Logical sectors are numbered from 0 to the highest sector number on the drive. These numbers are different from *physical* sector numbers, which are hardware-dependent. To calculate the number of logical sectors, take the drive size and divide by 512. For example, a 1.44 MB diskette has 2,880 sectors, calculated as 1,474,560 / 512.

Here is a disassembly of sector 0 read from a floppy disk, using Debug. This is commonly called the *boot record*. The boot record contains information about the disk, along with a short program that is responsible for loading the rest of the operating system when the computer starts up:

```
1F6E:0100 EB34          JMP     0136
...
1F6E:0136 FA            CLI
1F6E:0137 33C0          XOR     AX,AX
1F6E:0139 8ED0          MOV     SS,AX
1F6E:013B BC007C        MOV     SP,7C00
```

3.11 M (Move)

The M command copies a block of data from one memory location to another. The command format is:

```
M range address
```

Range consists of the starting and ending locations of the bytes to be copied. *Address* is the target location to which the data will be copied. All offsets are assumed to be from DS unless specified otherwise. Examples:

Example	**Description**
`M 100 105 110`	Move bytes in the range DS:100-105 to location DS:110.
`M CS:100 105 CS:110`	Same as above, except that all offsets are relative to the segment value in CS.

Sample String Move. The following example uses the M command to copy the string 'ABCDEF' from offset 100h to 106h. First, the string is stored at location 100h; then memory is dumped, showing the string. Next, we move (copy) the string to offset 106h and dump offsets 100h-10Bh:

```
-E 100 "ABCDEF"
-D 100 105
19EB:0100  41 42 43 44 45 46                      ABCDEF
-M 100 105 106
-D 100 10B
19EB:0100  41 42 43 44 45 46 41 42-43 44 45 46    ABCDEFABCDEF
```

3.12 N (Name)

The N command initializes a filename (and file control block) in memory before using the Load or Write commands. Comand format:

```
N  [d:][filename][.ext]
```

Example:

```
N b:myfile.dta
```

3.13 P (Proceed)

The P command executes one or more instructions or subroutines. Whereas the T (trace) command traces into subroutine calls, the P command simply executes subroutines. Also, LOOP instruction and string primitive instructions (SCAS, LODS, etc.) are executed completely up to the instruction that follows them. Command format:

```
P
P =address
P =address number
```

Examples are:

Example	**Description**
P =200	Execute a single instruction at CS:0200.
P =150 6	Execute 6 instructions starting at CS:0150.
P 5	Execute the next 5 instructions.

Example: Debugging a Loop. Let's look at an example where the P command steps through MOV and ADD instructions one at a time. When the P command reaches the LOOP instruction, however, the complete loop is executed five times:

```
-A 100
4A66:0100 mov cx,5      ; loop counter = 5
4A66:0103 mov ax,0
4A66:0106 add ax,cx
4A66:0108 loop 106      ; loop to location 0106h

-R
AX=000F  BX=0000  CX=0000  DX=0000  SP=FFEE  BP=0000  SI=0000  DI=0000
DS=4A66  ES=4A66  SS=4A66  CS=4A66  IP=0100   NV UP EI PL NZ NA PE NC
4A66:0100 B90500        MOV     CX,0005
-P
AX=000F  BX=0000  CX=0005  DX=0000  SP=FFEE  BP=0000  SI=0000  DI=0000
DS=4A66  ES=4A66  SS=4A66  CS=4A66  IP=0103   NV UP EI PL NZ NA PE NC
4A66:0103 B80000        MOV     AX,0000
-P
AX=0000  BX=0000  CX=0005  DX=0000  SP=FFEE  BP=0000  SI=0000  DI=0000
DS=4A66  ES=4A66  SS=4A66  CS=4A66  IP=0106   NV UP EI PL NZ NA PE NC
4A66:0106 01C8          ADD     AX,CX
-P
AX=0005  BX=0000  CX=0005  DX=0000  SP=FFEE  BP=0000  SI=0000  DI=0000
DS=4A66  ES=4A66  SS=4A66  CS=4A66  IP=0108   NV UP EI PL NZ NA PE NC
4A66:0108 E2FC          LOOP    0106
-P
AX=000F  BX=0000  CX=0000  DX=0000  SP=FFEE  BP=0000  SI=0000  DI=0000
DS=4A66  ES=4A66  SS=4A66  CS=4A66  IP=010A   NV UP EI PL NZ NA PE NC
```

3.14 Q (Quit)

The Q command quits Debug and returns to DOS.

3.15 R (Register)

The R command may be used to do any of the following: display the contents of one register, allowing it to be changed; display registers, flags, and the next instruction about to be executed; display all eight flag settings, allowing any or all of them to be changed. There are two command formats:

```
R
R register
```

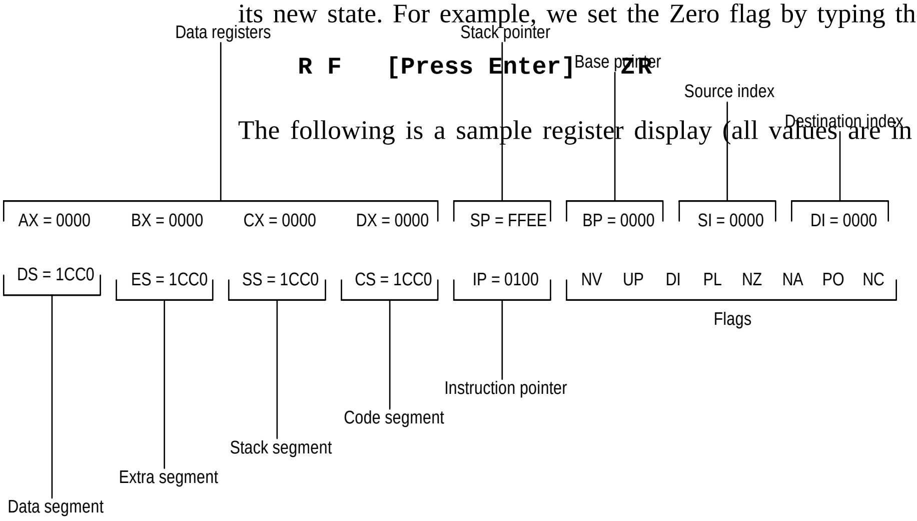
Here are some examples:

Example	**Description**
`R`	Display the contents of all registers.
`R IP`	Display the contents of IP and prompt for a new value.
`R CX`	Same (for the CX register).
`R F`	Display all flags and prompt for a new flag value.

Once the **R F** command has displayed the flags, you can change any single flag by typing its new state. For example, we set the Zero flag by typing the following two commands:

```
R F   [Press Enter]   ZR
```

The following is a sample register display (all values are in hexadecimal):

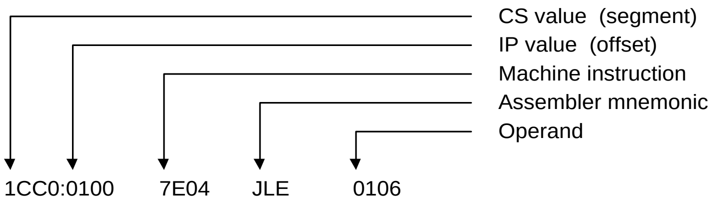


```
AX = 0000   BX = 0000   CX = 0000   DX = 0000   SP = FFEE   BP = 0000   SI = 0000   DI = 0000
DS = 1CC0   ES = 1CC0   SS = 1CC0   CS = 1CC0   IP = 0100   NV  UP  DI  PL  NZ  NA  PO  NC
```

The complete set of possible flag mnemonics in Debug (ordered from left to right) are as follows:

Set	***Clear***
OV = Overflow	NV = No Overflow
DN = Direction Down	UP = Direction Up
EI = Interrupts Enabled	DI = Interrupts Disabled
NG = Sign Flag negative	PL = Sign Flag positive
ZR = Zero	NZ = Not Zero
AC = Auxiliary Carry	NA = No Auxiliary Carry
PO = Odd Parity	PE = Even Parity
CY = Carry	NC = No Carry

The **R** command also displays the next instruction to be executed:



```
1CC0:0100    7E04    JLE      0106
```

3.16 S (Search)

The S command searches a range of addresses for a sequence of one or more bytes. The command format is:

`S` ***`range list`***

Here are some examples:

Example	**Comment**
`S 100 1000 0D`	Search DS:100 to DS:1000 for the value 0Dh.
`S 100 1000 CD,20`	Search for the sequence CD 20.
`S 100 9FFF "COPY"`	Search for the word "COPY".

3.17 T (Trace)

The T command executes one or more instructions starting at either the current CS:IP location or at an optional address. The contents of the registers are shown after each instruction is executed. The command formats are:

```
T
T count
T =address count
```

Where *count* is the number of instructions to trace, and *address* is the starting address for the trace. Examples:

Example	Description
`T`	Trace the next instruction.
`T 5`	Trace the next five instructions.
`T =105 10`	Trace 16 instructions starting at CS:105.

This command traces individual loop iterations, so you may want to use it to debug statements within a loop. The T command traces into procedure calls, whereas the P (*proceed*) command executes a called procedure in its entirety.

3.18 U (Unassemble)

The U command translates memory into assembly language mnemonics. This is also called *disassembling* memory. If you don't supply an address, Debug disassembles from the location where the last U command left off. If the command is used for the first time after loading Debug, memory is unassembled from location CS:100. Command formats are:

```
U
U startaddr
U startaddr endaddr
```

Where *startaddr* is the starting point and *endaddr* is the ending address. Examples are:

Example	Description
`U`	Disassemble the next 32 bytes.
`U 0`	Disassemble 32 bytes at CS:0.
`U 100 108`	Disassemble the bytes from CS:100 to CS:108.

3.19 W (Write)

The W command writes a block of memory to a file or to individual disk sectors. To write to a file, its name must first be initialized with the N command. (If the file was just loaded either on the DOS command line or with the Load command, you do not need to repeat the Name command.) The command format is identical to the L (load) command:

```
W
W address
W address drive firstsector number
```

Place the number of bytes to be written in BX:CX. If a file is 12345h bytes long, for example, BX and CX will contain the following values:

```
BX = 0001   CX = 2345
```

Here are a few examples:

Example	Description
N EXAMPLE.COM	Initialize the filename EXAMPLE.COM on the default drive.
R BX 0 R CX 20	Set the BX and CX registers to 00000020h, the length of the file.
W	Write 20h bytes to the file, starting at CS:100.
W 0	Write from location CS:0 to the file.
W	Write named file from location CS:0100.
W DS:0200	Write named file from location DS:0200.

The following commands are extremely dangerous to the data on your disk drive, because writing sectors can wipe out the disk's existing file system. Use them with extreme caution!

`W 100 2 A 5`	Write five sectors to drive C from location CS:100, starting at logical sector 0Ah.
`W 100 0 0 2`	Write two sectors to drive A from location CS:100, starting at logical sector number 0.

Table 2. Default Segments for Debug Commands.

Command	Description	Default Segment
A	Assemble	CS
D	Dump	DS
E	Enter	DS
F	Fill	DS
G	Go (execute)	CS
L	Load	CS
M	Move	DS
P	Procedure trace	CS
S	Search	DS
T	Trace	CS
U	Unassemble	CS
W	Write	CS

4 SEGMENT DEFAULTS

Debug recognizes the CS and DS registers as default segment registers when processing commands. Therefore, the value in CS or DS acts as a base location to which an offset value is added. Table 2 lists the default segment register for selected Debug commands.

The Assemble command, for example, assumes that CS is the default segment. If CS contains 1B00h, Debug begins assembling instructions at location 1B00:0100h when the following command is typed:

```
-A 100
```

The Dump command, on the other hand, assumes that DS is the default segment. If DS contains 1C20h, the following command dumps memory starting at 1C20:0300h:

```
-D 300
```

5 USING SCRIPT FILES WITH DEBUG

A major disadvantage of using Debug to assemble and run programs shows up when the programs must be edited. Inserting a new instruction often means retyping all subsequent instructions and recalculating the addresses of memory variables. There is an easier way: All of the commands and instructions may first be placed a text file, which we will call a script file. When Debug is run from DOS, you can use a redirection symbol (<) to tell it to read input from the *script file* instead of the console. For example, assume that a script file called input.txt contains the following lines:

```
a 100
mov ax,5
mov bx,10
add ax,bx
int 20
(blank line)
Q
```

(Always remember to put a Q on a line by itself at the end of your script file. The Q command returns to the DOS prompt.)

Debug can be executed, using the script file as input:

```
debug < input.txt
```

If you are running in a task-switching environment such as Windows, you can edit and save the script file with the Notepad editor or DOS Edit (in a separate window). Then switch back to a window in which you are running Debug. In this way, a program may be modified, saved, assembled, and traced within a few seconds. If you would like the output to be sent to a disk file or the printer, use redirection operators on the DOS command line:

```
debug < input.txt > prn            (printer)
debug < input.txt > output.txt     (disk file)
```