MS-DOS Debug 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] [number]
unassemble	U	[range]
write	W	[address] [drive] [firstsector] [number]

H o w t o u s e t h e
C O M M A N D S

P a r a m e t e r s

NOTE: Parameters listed in brackets ([ ]) are optional. Optional parameters usually indicate there are a number of different ways that a command can be used. I've listed the meanings of all the parameters here for you:

address - Memory location specified in hexadecimal. You can use either a simple Offset all by itself (in which case, the present CS 'Code Segment' will be assumed), or you can enter the full Segment:Offset location using either all hex numbers or substituting the name of a segment register for a number. Leading zeros are not required; thus 1F all by itself would be the location 'CS:001F' (CS meaning whatever the CS happened to be at the time you entered this). Examples:

100 DS:12 SS:0 198A:1234

range - Two hexadecimal addresses separated by a single space. They may be listed as either full Segment:Offset pairs or just an Offset alone (in which case, the Segment is assumed to be that of the present CS or "Code Segment"). NOTE: Some commands, such as Compare (C), may require that the second address be given only as an offset.

list - A string of Hexadecimal bytes separated by a space, or ASCII data enclosed within single or double quote marks. You can list any number of bytes from a single one up whatever number fits on

the line before having to press the Enter key. A single byte, such as 00 is most often used with the FILL (f) command whereas an ENTER (e) command will most likely have a string of many hex bytes or ASCII characters per line; for example:
e 100 31 C0 B4 09 BA 50 02 CD 21 B8 4C 00 CD 21
e 250 'This is an ASCII data string.$'

number - Remember that all numbers and values used in any DEBUG commands are understood as being Hexadecimal only! That includes the number of sectors in the LOAD or WRITE commands and even the number of instructions you want DEBUG to step through in the TRACE or PROCEED commands. It's all HEX all the time in here!

A Simple DEBUG Tutorial
Details of each Command

NOTE: In the Examples below, commands which are entered by a user are shown in **bold** type; data displayed in response by DEBUG is in normal type. DEBUG (from MS-DOS **5.0 or later**) will display the following *usage* message, if you enter **debug /?** at a DOS prompt:
C:\WINDOWS>**debug /?**
Runs Debug, a program testing and editing tool.
DEBUG [[drive:][path]filename [testfile-parameters]]
[drive:][path]filename Specifies the file you want to test.
testfile-parameters Specifies command-line information required by
the file you want to test.

Quit: Q
Immediately quits (exits) the Debug program! No questions ever asked... should be the first command you remember along with the "**?**" command.

Hex: H value1 value2
A very simple (add and subtract only) Hex calculator. Never forget that all numbers inside of DEBUG are always Hexadecimal. Enter two Hex values (no more than four digits each) and DEBUG shows **first** the SUM, then the DIFFERENCE of those values. Examples:
-h aaa 531 -h fff 3 -h dbf ace
0FDB 0579 1002 0FFC 188D 02F1

Dump: D [range]
D [address] [length]
Displays the contents of a block of memory. The Memory locations near the beginning of Segment **C000** (even under Windows 2000/XP) should display information about the kind of video card installed on your PC. This first example shows we have a Matrox card in this system.
Examples:

```
-d c000:0010
C000:0010 24 12 FF FF 00 00 00 00-60 00 00 00 00 20 49 42 $.......`.... IB
C000:0020 4D 20 43 4F 4D 50 41 54-49 42 4C 45 20 4D 41 54 M COMPATIBLE MAT
C000:0030 52 4F 58 2F 4D 47 41 2D-47 31 30 30 20 56 47 41 ROX/MGA-G100 VGA
C000:0040 2F 56 42 45 20 42 49 4F-53 20 28 56 31 2E 32 20 /VBE BIOS (V1.2
C000:0050 29 00 87 DB 87 DB 87 DB-87 DB 87 DB 87 DB 87 DB )...............
C000:0060 50 43 49 52 2B 10 01 10-00 00 18 00 00 00 00 03 PCIR+...........
C000:0070 40 00 12 10 00 80 00 00-38 37 34 2D 32 00 FF FF @.......874-2...
C000:0080 E8 26 56 8B D8 E8 C6 56-74 22 8C C8 3D 00 C0 74 .&V....Vt"..=..t
-
```

Search: S range list
Searches within a range of addresses for a pattern of one or more byte values given in a list. The list can be comprised of numbers *or character strings enclosed by matching single or double quote marks.* Examples:

```
-s fe00:0 ffff "BIOS"
FE00:0021
FE00:006F
```

```
-d fe00:0
FE00:0000 41 77 61 72 64 20 53 6F-66 74 77 61 72 65 49 42 Award SoftwareIB
FE00:0010 4D 20 43 4F 4D 50 41 54-49 42 4C 45 20 34 38 36 M COMPATIBLE 486
FE00:0020 20 42 49 4F 53 20 43 4F-50 59 52 49 47 48 54 20 BIOS COPYRIGHT
FE00:0030 41 77 61 72 64 20 53 6F-66 74 77 61 72 65 20 49 Award Software I
FE00:0040 6E 63 2E 6F 66 74 77 61-72 65 20 49 6E 63 2E 20 nc.oftware Inc.
FE00:0050 41 77 03 0C 04 01 01 6F-66 74 77 E9 12 14 20 43 Aw.....oftw... C
FE00:0060 1B 41 77 61 72 64 20 4D-6F 64 75 6C 61 72 20 42 .Award Modular B
FE00:0070 49 4F 53 20 76 34 2E 35-31 50 47 00 DB 32 EC 33 IOS v4.51PG..2.3
```

Compare: C range address

Compares two blocks of memory. If there are no differences, then DEBUG simply displays another prompt (-). Here's an example of what happens when there *are* differences:

```
-c 140 148 340
127D:0143 30 6D 127D:0343
127D:0146 10 63 127D:0346
127D:0148 49 30 127D:0348
```

The bytes at locations 140 through 148 are being compared to those at 340 (through 348, *implied*); the bytes are displayed side by side for those which are different (with their exact locations, including the segment, on either side of them).

Fill: F range list

This command can also be used to *clear* large areas of Memory as well as *filling* smaller areas with a continuously repeating phrase or single byte. Examples:

```
-f 100 12f 'BUFFER'
-d 100 12f
xxxx:0100 42 55 46 46 45 52 42 55-46 46 45 52 42 55 46 46 BUFFERBUFFERBUFF
xxxx:0110 45 52 42 55 46 46 45 52-42 55 46 46 45 52 42 55 ERBUFFERBUFFERBU
xxxx:0120 46 46 45 52 42 55 46 46-45 52 42 55 46 46 45 52 FFERBUFFERBUFFER
```

Enter: E address [list]

Used to enter data or instructions (as *machine code*) directly into Memory locations. Example: First we'll change a single byte at location CS:**FFCB** from whatever it was before to **D2**

```
-e ffcb d2
```

The next two examples show that either single(') or double(") quote marks are acceptable for entering ASCII data. By allowing both forms, you can include the other type of quote mark within your entry string:

```
-e 200 'An "ASCII-Z string" is always followed by '
-e 22a "a zero-byte ('00h')." 00
```

Go: G [=address] [addresses]

Go is used to run a program and set *breakpoints* in the program's code. As we saw in an Example for the ENTER command, the '=address' option is used to tell DEBUG a starting location. If you use 'g' all by itself, execution will begin at whatever location is pointed to by the CS:IP registers. Optional *breakpoints* (meaning the program will HALT before executing the code at any of these locations) of up to any ten addresses may be set by simply listing them on the command line.

Requirements: Breakpoints can only be set at an address containing the first byte of a valid **8088/8086 Opcode**. So don't be surprised if picking some arbitrary address never halts the program; especially if you're trying to DEBUG a program containing opcodes DEBUG can't understand (that's anything 'requiring' a CPU above an 8088/8086)!

CAUTION: DEBUG replaces the original instructions of any listed breakpoint addresses with **CC**h (an INT 3). The instructions at these locations are restored to their originals ONLY if one of the breakpoints is encountered... If DEBUG does not HALT on any breakpoint, then all your breakpoints are still enabled! So, don't ever save the code as is, unless you're sure that DEBUG has hit one of your breakpoints! (Saving to a backup copy *before using breakpoints* is often a better way.)

Assemble: A [address]

Creates machine executable code in memory beginning at CS:0100 (or the specified address) from the 8086/8088 (and 8087) Assembly Language instructions which are entered. Although no Macro instructions nor labels are recognized, you can use the *pseudo-instructions* 'DB' and 'DW' (so you can use the DB opcode to enter ASCII data like this: DB 'This is a string',0D,0A).

The 'A' command remembers the last location where any data was assembled, so successive 'A' commands (when no address is specified) will always begin at the next address in the chain of assembled instructions. This aspect of the command is similar to the Dump command which remembers the location of its last dump (if no new address is specified).

The assembly process will stop ***after*** you ENTER an empty line.

Example (you enter the characters in bold type):

```
-a
xxxx:0100 jmp 126
xxxx:0102 db 0d,0a,'This is my first DEBUG program!'
xxxx:0123 db 0d,0a,'$'
xxxx:0126 xor ax,ax
xxxx:0128 mov ah,9
xxxx:012A mov dx,102
xxxx:012D int 21
xxxx:012F mov ax,4c
xxxx:0132 int 21
xxxx:0134
-g =100
This is my first DEBUG program!
Program terminated normally
-
```

Unassemble: U [range]

Disassembles machine instructions into 8086 Assembly code. **Without the optional [range]**, it uses **Offset 100** as its starting point, disassembles about 32 bytes and then remembers the next byte it should start with if the command is used again. (The word 'about' was used above, because it may be necessary to finish with an odd-number of bytes greater than 32, depending upon the last type of instruction DEBUG has to disassemble.

NOTE: The user must decide whether the bytes that DEBUG disassembles are all 8086 **instructions**, just **data** or any of the newer x86 instructions (such as those for the 80286, 80386 on up to the latest CPU from Intel; which are all beyond the ability of DEBUG to understand)!

Example:

```
-u 126 133
xxxx:0126 31C0 XOR AX,AX
xxxx:0128 B409 MOV AH,09
xxxx:012A BA0201 MOV DX,0102
xxxx:012D CD21 INT 21
xxxx:012F B84C00 MOV AX,004C
xxxx:0132 CD21 INT 21
-
```

Input: I port

The use of I/O commands while running Windows™9x/Me is *just plain* unreliable! This is especially true when trying to directly access hard disks! Under Win NT/2000/XP, the I/O commands are only an ***emulation***; so don't trust them. Though the example below still works under Win2000/XP, it's most likely using some WinAPI code to show what's in the Windows clock area; *not* directly from an RTC chip.

Long ago (when DOS was the only OS for PCs), there were dozens of **BASIC** programs that used I/O commands for handling tasks through parallel and serial ports (e.g., to change the font used by a printer or values in a modem's control registers). Under real DOS, they can still be used for direct communications with keyboards or a floppy drive's control chips along with many other hardware devices.

Here's an example of how to read the hours and minutes from a computer's "real time clock" (RTC):

-o 70 04 <-- Check the **hours**.

-i 71

18 <----- 18 hours (or 6 p.m.)

-o 70 02 <-- Check the **minutes**.
-i 71
52 <----- 52 minutes
Output: O port byte
See comments under the **Input** command.
Load:
L [address] [drive] [firstsector] [number] ***or program!*** **(See the N command for more on this)**
This command will LOAD the selected number of sectors from any disk's Logical Drive under the control of MS-DOS or Windows into Memory. The **address** is the location in Memory the data will be copied to (use only 4 hex digits to keep it within the memory allocated to DEBUG), the **drive** number is mapped as: 0=A:, 1=B:, 2=C:, etc., **firstsector** counts from ZERO to the largest sector in the volume *and finally* **number** specifies **in hexadecimal** the ***total*** **number** of sectors that will be copied into Memory (so a floppy disk with 0 through 2,879 sectors would be: 0 through **B3F** in Hex).
Move: M range address
This command should really be called: COPY (not Move) as it actually *copies* all the bytes from within the specified **range** to a new **address**.
Examples:
```
1) -m 7c00 7cff 600
```
Copies all the bytes between Offset 7C00 and 7CFF (inclusive) to Offset 0600 and following...
```
2) -m 100 2ff 70
```
This second example shows that it's very easy to ***overwrite*** most of the source you're copying from using the *Move* command. Apparently, DEBUG stores the source bytes elsewhere before writing them; otherwise, this example would cause a problem when it started overwriting what it hadn't copied yet! This copies the 512 bytes between Offsets 100h and 2FFh (inclusive) to **Offset 0070** overwriting the first 368 bytes in the process.
Name: N [pathname] [arglist]
This command can be used to load files into DEBUG's Memory *after* you have started the program, but it's main function is to create a new file under control of the Operating System which DEBUG can WRITE data to.
Normally, when you want to 'debug' a file, you'd start DEBUG with a command like this: C:\WINDOWS>**debug test.com** . But it's also possible to load a file into DEBUG's Memory from within DEBUG itself by using the 'N' command *and then the* 'L' command **(**with **no** parameters**)** like this:
```
-n c:\temp\test.com
-l
```
which will load the file test.com into DEBUG's Memory starting at location CS:0100 **(**you cannot specify any other location when using the L command like this!**)**.
The 'N' command makes it quite easy to save data or an Assembly program created in DEBUG to a file on your hard drive!
For example, these commands (**in bold**; along with DEBUG's reponses):

```
-n c:\temp\doswinok.com
-a 100
cs:0100 jmp 138
cs:0102 db 0d,0a,"It's OK to run this "
cs:0118 db "program under DOS or Windows!"
cs:0135 db 0d,0a,24
cs:0138 mov dx,102
cs:013B mov ah,9
cs:013D int 21
cs:013F mov ax,4c01
cs:0142 int 21
cs:0144
```

```
-rcx
CX 0000
:44
-w
Writing 00044 bytes [ 68 bytes in decimal ]
-q
```

will create a 68-byte file called DOSWINOK.COM in the C:\TEMP folder; *even* when running DEBUG in a DOS-window. The file names, however, are still limited to DOS's **eight** characters *plus* **three** for the extension (an *8.3 filename* as it's often called)!
Homework: Follow the steps above to Assemble and save this program under DEBUG, then use DEBUG to ***debug it!*** Use the P(roceed) command to **step through** most of the instructions, ***since*** this will keep you from accidentally *stepping into* an **INT**(errupt) instruction! If you ever do use the T(race) command on an INT, you'll end up inside nests of **BIOS** routines which often *crashes* DEBUG!

Register: R [register]

Entering ' r ' all by itself will display *all* of the 8086 register's contents *and* the next instruction which the IP register points to in both machine code and an unassembled (Assembly Language) form. For example, if you start DEBUG in a Windows 95B DOS-box with the command line:
>**debug c:\windows\command\choice.com**
and then enter an ' r ' at the first DEBUG prompt, DEBUG will display someting similar to this:

```
AX=0000 BX=0000 CX=1437 DX=0000 SP=FFFE BP=0000 SI=0000 DI=0000
DS=0ED8 ES=0ED8 SS=0ED8 CS=0ED8 IP=0100 NV UP EI PL NZ NA PO NC
0ED8:0100 E90E01 JMP 0211
```

Flags Register

Flag Name	Set	Clear
Overflow(yes/no)	OV	NV
Direction(increment/decrement)	DN	UP
Interrupt(enable/disable)	EI	DI
Sign(negative/positive)	NG	PL
Zero(yes/no)	ZR	NZ
Auxiliary carry(yes/no)	AC	NA
Parity(even/odd)	PE	PO
Carry(yes/no)	CY	NC

Trace: T [=address] [number]

The T command is used to trace (step through) CPU instructions one at a time. If you enter the T command all by itself, it will step through only ONE instruction beginning at the location specified by your CS:IP registers, halt program execution and then display all the CPU registers plus an unassembled version of the next instruction to be executed; this is the 'default' mode of the TRACE command. Say, however, you wanted DEBUG to trace and execute seven instructions beginning at address CS:0205; to do so, you would enter:
-t =205 7
Remember that the value for the number of instructions to execute must be given in hexadecimal just as all other values used in DEUBG. (Since the T command uses the "hardware trace mode" of the CPU, it's possible to step through instructions in a ROM - Read Only Memory - chip.)

Proceed: P [=address] [number]

Proceed acts exactly the same as Debug's T (Trace) command for most types of instructions... EXCEPT: *Proceed* will immediately execute ALL the instructions (rather than stepping through each one) inside any Subroutine CALL, a LOOP, a REPeated string instruction or any software INTerrupts. This means that you do not have to single-step through any of the code contained in a Subroutine or INT call if you use the Proceed (P) command. This means *Proceed* will be the command you use most often to debug programs, and Trace will only be used to step into a Subroutine or possibly check the logic of the first few iterations of a LOOP or REP instruction.

Write:

W [address] [drive] [firstsector] [number]

W A R N I N G

Do NOT *experiment* with the **W - write** command in DEBUG. It can be used effectively to create new files on your hard drive, but only if you use it properly. Trying to write directly to a sector on a hard disk would very RARELY be considered proper use of this command!

Trying to write directly to a hard disk using sector numbers will most likely result in loss of data or even a corrupted hard drive!

The WRITE (W) command is often used to save a program to your hard disk from within DEBUG. But the only safe way to do so, especially under Windows, is by allowing the OS to decide where to physically create that file on the disk. This is done by first using the Name (N) command to set up an *optional path* and **filename** for the new file (or to overwrite one that already exists). DEBUG will automatically begin saving program or data bytes from **Offset 0100** of the 64 KiB Segment that the OS allocated for it. The only other requirement is to set the size of the file you wish to write by placing the total number of bytes in the **combined BX** and **CX registers*** before executing the WRITE command. The Register command is used to change the value in **the CX register.**