A Guide to DEBUG

(The Microsoft® DEBUG.EXE Program)

Daniel B. Sedory

This page may be freely copied for PERSONAL use ONLY ! (It may NOT be used for ANY other purpose unless you have first contacted and received permission from the author !)

A History of MS-DEBUG

Beginnings

In 1980, Tim Paterson began working on a 16-bit OS for the 8086 S-100 Bus card he had designed for SCP (Seattle Computer Products) the previous year. To help in getting QDOS (later called 86-DOS) to work correctly, Tim created a debugger in a ROM chip; the code for that ROM version was released into the Public Domain. Later, Tim adapted the code to run as a .COM program under QDOS, and also added the ability to disassemble 8086 machine code. In the meantime, Microsoft® had been busy purchasing the rights to sell Tim's QDOS to IBM® for their 'secret' PC project. Tim was then hired by Microsoft as the primary author of their first OS. When he completed his work on IBM's Personal Computer™ DOS 1.00 in 1981, his DEBUG.COM utility was included with it. All the functionality that Tim put into DEBUG is still there and little has been added to it (the major exception being the Assemble command; added under DOS 2.0).

[Thanks go to Tim Paterson himself for reviewing this perspective on DEBUG's beginnings.]

Changes in MS-DEBUG

With the release of DOS 2.0, DEBUG gained the ability to assemble instructions directly into machine code (the A command). This is one of the most important commands for many of its users. Though lacking much of the functionality of a stand-alone Assembler, e.g., all Jumps must be to hexadecimal addresses (no labels can be used), many useful .COM programs have been assembled with this command. Under DOS 3.0, the P (Proceed) command was added, so DEBUG could quickly execute subroutines; at the same time, it became possible to attempt stepping through Interrupts with the T (Trace) command. When DOS made EMS (Expanded Memory) functions available under DOS 4.0, the four commands xa, xd, xm and xs were also added to DEBUG. It appears they were rarely, if ever used though, even by programmers. For most of us, the only noticeable change in DEBUG was the addition of the help command (type a '?' while inside DEBUG) under DOS 5.0; when all DOS commands finally got the /? command-line switch.

DEBUG's code went through a number of changes (and 'bug fixes' too) over the years! Some of these internal changes were related to DOS system calls and screen output, then there was the change in file type from a .COM to an .EXE program under DOS 5.0. But in spite of all those changes and others which followed, DEBUG has never had an official revision since 2.40 (those digits have been embedded inside all versions of DEBUG since DOS 3.0). We can only guess about the real reasons that Microsoft® never updated DEBUG to handle instructions beyond those of the Intel® 8086/8087/8088 processors. Microsoft® did create their own Assembler (MASM), 'C' compiler and Debugger (CodeView); which you could use too, if you were willing to pay extra, so that could have been one of their reasons. Rather than using MASM and CodeView, many opted for the less expensive Borland® assembler (TASM) and Turbo™ Debugger when they appeared, or some other commercial product. However, users and students alike can still learn a great deal about Assembly language by using DEBUG.

DEBUG under Windows® 9x/Me

The internal structure of these Windows® versions of DEBUG appear much different than any previous DOS forms; at least at first glance. Though it seems a great deal has changed, you'll still find the phrase "Vers 2.40" but in a different location. Windows® itself went through a lot of changes during this period, such as being able to handle a new file system, FAT32, and larger drives. But

without access to its source code, we can't be sure if there were any major differences in DEBUG. The changes might be due to something as simple as just reorganizing the error messages in the source code and/or using a new Assembler/Linker.

DEBUG had always been an effective tool in the hands of any Batch programmer, but some time after the introduction of Windows™ 95 and especially with Windows™ 98, that effectiveness was diminished when its I/O commands became unreliable! Whether due to a 'bug' in DEBUG itself or in Windows®, the fact is that I/O commands under Windows™ 9x/Me cannot be relied upon for direct access to a hard drive! If you run our ATA drive ID script under Win9x/Me, the data you get back is as interesting as it is disturbing: It appears that every other byte is still correct! So, one has to wonder what the cause of this problem might be.

DEBUG under Windows® NT/2000/XP/2003

The DEBUG program included with Windows® NT/2000/XP/2003 appears to operate the same as it did under DOS 5.0, but with two major exceptions:

1. DEBUG is no longer allowed to load from or write to any logical HDD sectors; only named files can still be read from or written to under an NT-type OS. It can, however, still access diskette sectors in the A:\ or B:\ drives with the L and W commands, but only if those diskettes contain a file system the OS can recognize! (See the L command in the Tutorial section for more information.)

   [Note: DEBUG has never been able to directly access areas of an HDD outside of its drive volumes; such as an Extended partition table or even the MBR sector! However, DEBUG can be used to access such data by programming it to run INT13 commands or using a script file under DOS (e.g., our old CopyMBR script).]

2. The I and O commands are essentially useless, since the program's interface with the rest of the system is only being emulated under these versions of Windows® rather than having any direct access to the hardware. This was already true to varying degrees under previous versions of Windows®.

This may surprise you: We purposely mentioned the DOS 5.0 version of DEBUG here, since the DEBUG.EXE file included with Windows® XP (and every other version of the NT OS series) is exactly the same program file created for MS-DOS 5.0 (md5sum = c17afa0aad78c621f818dd6729572c48).

DEBUG was only one of a small handful of DOS 5.0 programs that didn't require any changes to run under an NT operating system. It's almost ironic that another of those few programs is EDLIN, a line editor disliked by most DOS users. Though EDLIN was also created by Tim Paterson, he did so in just two weeks and was shocked when he heard IBM had actually included it in their PC-DOS 1.00 release! No doubt he wished it had been replaced by something better way back in 1981. It wasn't until the release of DOS 5.0, that EDLIN was effectively replaced by Microsoft's EDIT.COM program (v 1.0, 1991; QBASIC must be present, or it's useless); EDLIN was, however, still retained, to be 'backwards compatible' with various 3rd-party Batch files. Though NOTEPAD or more advanced editors are available under Windows™, you can still use the 1995 standalone version of EDIT (v 2.0.026) at Command Line prompts in Windows™ XP; its menus will even respond to mouse clicks.

Summary

Though created at the beginning of the 16-bit processor era (before the 80286 existed), more recent versions of DEBUG (such as those found inside a Windows™ Me or 98SE Emergency Boot Diskette's EBD.CAB file) are still useful to PC techs for direct access to certain memory locations on present-day

systems (an Intel® Pentium™ IV, for example). DEBUG can also be quite useful for educational purposes. And even for debugging the Assembly code that is required during the boot process: The software that checks the Partition Table on hard disks and loads OS Boot Sectors into Memory. Unfortunately, many Boot Managers and recent MBR sectors now use instructions requiring a 386 (or even 486) class CPU to function, making it difficult to use DEBUG for such a purpose. Because of the backward compatibility of most Intel® processors, and the fact that it was included with Microsoft® Windows™ XP and 2003, DEBUG has had a much longer life span than ever expected. Though the ITANIUM™ CPU was not x86-compatible, the AMD64 was. In 2005, Intel® made the so-called "x64-based" CPUs that were once again x86-compatible. So, DEBUG still continues to find some use on 64-bit computers, even my new Intel® Core™ 2 Quad (4 processors in one) machine.

The Limitations of MS-DEBUG

DEBUG was originally designed to work with .COM programs having a maximum size of only 65,280 bytes [(64 x 1024) - 256] or less; how much less, depended upon the maximum number of bytes the program had to place on the Stack at the same time. The subtraction of 256 bytes is necessary since DEBUG often uses the area from offset 00 through FF hex for some internal data such as the name of the file that was loaded. Remember, true .COM programs by definition are supposed to fit inside a single Segment of memory (only 64 KiB).

Even when running MS-DEBUG under the latest Windows® OS, since it's still an old 16-bit DOS application, you can only open files whose names have been saved in the 8.3 DOS convention; i.e., up to 11 characters total, using no more than 8 DOS characters for the name and 3 for the extension.
As early as DOS 1.10, DEBUG was able to load files larger than 64 KiB. Basically, how large a file that DEBUG can safely use without error depends on the amount of available memory and the way the OS handles memory management. We'll say more about this below.

A. When DEBUG starts with no command-line parameters, it:

1) Allocates all 64 KiB of the first free Memory Segment.

2) The Segment registers, CS, DS, ES and SS are all set to the value of that 64 KiB Segment's location (CS=DS=ES=SS=Segment Location).

3) The Instruction Pointer (IP) is set to cs:0100 and the Stack Pointer (SP) is set to ss:FFEE (under DOS 3.0 or above).

4) The registers, AX, BX, CX, DX, BP, SI and DI are cleared to zero along with the flag bits in the Flag Register; with one exception: The Interrupts Flag is set to Enable Interrupts. (See the Appendix, The 8086 CPU Registers for more information.)

B. When DEBUG starts with a filename (other than an .EXE), it:

1) Allocates at least 64 KiB of the first free Memory Segment for debugging programs or examining files specified on the command line. [Note: Ever since DOS version 1.10, DEBUG has had the ability to load (and save) files larger than 64 KiB. Just how large a file it can handle, depends upon both the OS and available memory. But before you ever consider using DEBUG to save some large file you want to edit, you should know the amount of memory it can use is limited to what's available in CONVENTIONAL MEMORY only! And remember that just because your system can debug a certain file, doesn't mean someone else's will be able to.]

2) The Segment registers, CS, DS, ES and SS are all set to the value of the first 64 KiB Segment's location (CS=DS=ES=SS=Segment Location); for a file that's larger than 64KiB, you'll have to set different segment values to access all the bytes loaded into memory beyond the first 64 KiB.

3) The Instruction Pointer (IP) is set to cs:0100 and the Stack Pointer (SP) is set to ss:FFFE (version 3.0+). [Note: This is not the same as the ss:FFEE in A. 3) above; there's a 16 byte difference.]

4) Most of the registers follow the same pattern as above, except for the CX and sometimes BX registers: The size of the file will be placed into the linear combination of the BX and CX registers; for files less than 64 KiB - 256 bytes, only CX is used. Example: If you can load a file of 360,247 bytes, then BX=0005 and CX=7F37 (57F37 hex = 360,247). If you load a file of exactly 65,536 bytes from a prompt, these registers will be: BX=0001, CX=0000. But due to the automatic 100h load offset, the file's last 256 bytes will have been loaded at the beginning of the next 64 KiB segment.

Remember: The Segment assigned to DEBUG, depends on the amount of memory in use, not the total memory available. So, the same DOS machine, whether it has 16 or even 4096 MiB of memory, will generally load DEBUG into the same Segment; unless a "terminate and stay resident" program is using that memory, or memory was not properly deallocated prior to running DEBUG.

Using DEBUG with .EXE Files

Any version of DEBUG from DOS 2.0 or higher, makes use of the operating system's EXEC function which means that it's possible for you to perform a limited amount of debugging on an .EXE program. However, DEBUG can never be used to save either an .EXE or a .HEX file to disk, since both of these file types contain extra data that DEBUG was never programmed to create after EXEC removed such data! It is quite possible though, to change the extension of an .EXE file, for example to .BIN, so DEBUG can be used to edit such a file, then change it back to an .EXE extension afterwards. Normally we'd recommend using a Hex editor instead, but would like to point out that DEBUG could be used with Batch files and scripts to carry out such edits automatically; taking the place of a Patch program.

One of the simplest .EXE programs you can run under DEBUG is the so-called DOS "Stub" found inside many Windows® executables. You can follow along as we examine one of these here!

Special Memory Locations in MS-DEBUG

There will always be some code and data placed within the first 256 bytes of the Segment for DEBUG's own use. And although DEBUG often functions as expected if you zero-out this area, there may be some cases where you wouldn't want to alter its contents. The code bytes are simple and always found in the same locations: The first two bytes of this area ("CD 20") are machine code for the DOS interrupt: INT 20. The bytes at offsets 50h and 51h ("CD 21") form an INT 21, and the byte "CB" at offset 52h is a RETF instruction.

When booting from an MS-DOS 6.22 upgrade install disk, this area will appear as follows (offsets 90h - FFh were all zero bytes):

```
A:\>debug
-d 0 8f
1787:0000  CD 20 C0 9F 00 9A EE FE-1D F0 4F 03 EB 11 8A 03   . ........O.....
1787:0010  EB 11 17 03 EB 11 38 0E-01 01 01 00 02 FF FF FF   ......8.........
1787:0020  FF FF FF FF FF FF FF FF-FF FF FF FF 5F 0F 4E 01   ............_.N.
1787:0030  AB 16 14 00 18 00 87 17-FF FF FF FF 00 00 00 00   ................
1787:0040  06 16 00 00 00 00 00 00-00 00 00 00 00 00 00 00   ................
1787:0050  CD 21 CB 00 00 00 00 00-00 00 00 00 00 20 20 20   .!...........
1787:0060  20 20 20 20 20 20 20 20-00 00 00 00 00 20 20 20           .....
1787:0070  20 20 20 20 20 20 20 20-00 00 00 00 00 00 00 00           ........
1787:0080  00 0D 75 73 0D 0D 00 00-00 00 00 00 00 00 00 00   ..us...........
```

When running DEBUG without a filename, whatever appeared on the previous command line, except for the command itself, will be displayed in the bytes at offsets 82h and following. These are often referred to as DOS switches or parameters. Thus, the "us" in the display above was from the command "keyb us" in the DOS 6.22 install disk's AUTOEXEC.BAT file. And if we had run the command "dir /w" before executing DEBUG, a "/w" would have appeared here instead.

Note: Successive uses of DOS parameters are never cleared from memory, only overwritten.

So, many characters of a very long parameter string will often remain intact, and as a consequence, will be copied to the bytes at offsets 82h through FFh each time DEBUG is run.
One of these data locations (the Word at offsets 36h - 37h) clearly saves the Segment assigned to DEBUG for our use. However, if we go on to load the file edit.com and dump the beginning of the Segment again, we'll find the Segment value itself has changed from 1787h to 1798h (a difference of 11h or 17 paragraphs, amounting to 256 + 16 = 272 bytes):

```
-n edit.com
-l
-d 0 8f
1798:0000  CD 20 C0 9F 00 9A F0 FE-1D F0 4F 03 EB 11 8A 03   . ........O.....
1798:0010  EB 11 17 03 EB 11 DA 11-01 01 01 00 02 FF FF FF   ................
1798:0020  FF FF FF FF FF FF FF FF-FF FF FF FF 75 0F E8 49   ............u..I
1798:0030  EB 11 14 00 18 00 98 17-FF FF FF FF 00 00 00 00   ................
1798:0040  06 16 00 00 00 00 00 00-00 00 00 00 00 00 00 00   ................
1798:0050  CD 21 CB 00 00 00 00 00-00 00 00 00 00 45 44 49   .!...........EDI
1798:0060  54 20 20 20 20 43 4F 4D-00 00 00 00 00 20 20 20   T    COM.....
1798:0070  20 20 20 20 20 20 20 20-00 00 00 00 00 00 00 00           ........
1798:0080  09 20 45 44 49 54 2E 43-4F 4D 0D 00 00 00 00 00   . EDIT.COM......
```

At first, we were unsure why DEBUG was doing this, but knew it had nothing to do with the size of this program, which is only 413 bytes. Instead it's simply because this is a "program" (EDIT.COM) rather than some other type of file. DEBUG did not do this when loading much larger files of other types, but did so again when loading any of the disk's *.EXE programs. We then confirmed a similar change when loading *.COM or *.EXE programs into DEBUG on an XP machine, but the change there was larger; it added up to 91 paragraphs! After more experiments, using the SET and PATH commands, we discovered DEBUG had some need to

load a copy of the DOS "environment variables" between its initial "data area" and a new 256-byte data area it creates when debugging only DOS "program" files. Even when no PATH or environmental variables exist, DEBUG still needs to create a new "data area" for *.COM or *.EXE files.

When running DEBUG in a Windows® DOS-box (under CMD.exe), dumping its first 256 bytes will almost always show the same fragmented string (shown below in white text). The characters are the remains of the Ntvdm program (which starts as soon as any 16-bit command is run) quickly reading one line at a time from the file AUTOEXEC.NT (located in the C:\WINDOWS\system32 folder), into the same area of memory where command line parameters are stored. The longest line in that file, including its trailing 0Dh (carriage return) byte, is successively overwritten by shorter lines in the file until the process results in what's copied to offsets 82h through CEh of DEBUG's Segment:

```
C:\>debug
-d 0 cf
0B20:0000  CD 20 FF 9F 00 9A EE FE-1D F0 4F 03 84 05 8A 03   . ........O.....
0B20:0010  84 05 17 03 84 05 25 04-01 01 01 00 02 FF FF FF   ......%.........
0B20:0020  FF FF FF FF FF FF FF FF-FF FF FF FF 28 05 4E 01   ............(.N.
0B20:0030  44 0A 14 00 18 00 20 0B-FF FF FF FF 00 00 00 00   D..... .........
0B20:0040  05 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00   ................
0B20:0050  CD 21 CB 00 00 00 00 00-00 00 00 00 00 20 20 20   .!...........
0B20:0060  20 20 20 20 20 20 20 20-00 00 00 00 00 20 20 20           .....
0B20:0070  20 20 20 20 20 20 20 20-00 00 00 00 00 00 00 00           ........
0B20:0080  00 0D 20 20 20 53 45 54-20 42 4C 41 53 54 45 52   ..   SET BLASTER
0B20:0090  3D 41 30 0D 64 64 72 65-73 73 2E 20 20 46 6F 72   =A0.ddress.  For
0B20:00A0  20 65 78 61 6D 70 6C 65-3A 0D 20 6F 6E 20 4E 54    example:. on NT
0B20:00B0  56 44 4D 2C 20 73 70 65-63 69 66 79 20 61 6E 20   VDM, specify an
0B20:00C0  69 6E 76 61 6C 69 64 0D-20 6F 6E 6C 79 2E 0D 00   invalid. only...
```

None of the line feeds (0Ah) at the end of each line in AUTOEXEC.NT will ever appear here, because the carriage returns (0Dh) just preceding them send the cursor to the start of the line each time they're encountered, and whatever comes before the first space character (20h) in every line does not get copied; which is why the "REM" of the last three lines doesn't appear here either.

The byte at offset 81h is always 0Dh; even if the file AUTOEXEC.NT contains a single byte of any value.

Note: If you rename or delete AUTOEXEC.NT, you will not be allowed to run DEBUG (nor any other 16-bit program; all of which must run under Ntvdm). You can, however, save a copy of AUTOEXEC.NT, then edit it to see how your changes affect what's copied into DEBUG. You may reduce its size to just a single byte. But in order to see anything other than zero bytes in offsets 82h and following, at least one space byte (20h) must be placed between a non-space byte at the beginning of a line and whatever you'd like to have displayed. If the file contains only the 3 bytes: "T", space and "S", then offsets 82h and 83h would be an "S" followed by 0Dh.

DEBUG's "Dynamic Stack" Area

This section is presently a "Work in Progress", but if you happen to see this before it's finished, can you guess what it's about?

Have you found a "Bug" in DEBUG?

Although almost all the code used by programmers performs exactly as expected; once they've eliminated their own errors in logic that is, occasionally it will produce surprising results because they didn't dig deep enough into the fine print of the user manuals for a PC's processor. Professional programmers will always test their code in as many ways as reasonably possible, but studying the processor's programming notes; especially sections pertinent to any of their tasks, should be high on their list! NOTE: If you want to be a much better hacker, the example presented here might cause you to delve into Intel's detailed notes on how their CPUs handle various instructions.

Have you ever encountered two distinct lines of Assembly instructions that DEBUG always steps through without ever stopping at the second line? The following is just one of MANY repeatable examples we could list here. Open any instance of DEBUG (DOS or Windows; any version), and copy and paste the following E (Enter) command at its (-) prompt:

```
e 100 8C C8 8E D0 B0 74 90 90
```

After entering "u 100 107" it should disassemble to:

```
xxxx:0100 8CC8     MOV   AX,CS  <- Keep Stack in CS Segment.
xxxx:0102 8ED0     MOV   SS,AX  <- The key instruction!
xxxx:0104 B074     MOV   AL,73  <- Could be almost anything.
xxxx:0106 90       NOP
xxxx:0107 90       NOP
```

Now enter an "r" at the prompt and try to single step (t) through the code. As soon as you enter the t command at offset 0102h, you'll wind up at offset 0106h; every time! Is this some "bug" that was never dealt with? The instruction at offsets 0104h ff. could be almost anything; any 1-, 2-, 3- or even 4-byte machine code will do; we purposely picked one that would alter a register's contents (AL in this case) so you could see it had been executed by the CPU.

This effect will always be observed no matter what version of DEBUG you run it under; MS-DOS 7.1, 5.0 or all the way back to the first version of DEBUG under IBM PC DOS 1.0 (no guarantee it would act the same if you ran it on an original PC though; we only have an 80486 and later for testing). However, if you expand your research to include other debugging tools, you'll soon realize the chances of every version of two or more tools having the same "bug" are... well, way too coincidental. So, why does this code affect a debugger's interrupt abilities?

If you haven't already reached for your Intel® Instruction Set Reference... What's that? You don't have one! Well, you'd better search for the keywords: Intel, IA32, Software, Instruction and at least download a PDF digital copy of the Instruction Set Reference! (Usually found as two separate files titled: Volume 2A: Instruction Set Reference, A-M and Volume 2B: Instruction Set Reference, N-Z). In my January 2006 copy under "MOV—Move", I found:

"Loading the SS register with a MOV instruction inhibits all interrupts until after the execution of the next instruction. This operation allows a stack pointer to be loaded into the ESP register with the next instruction (MOV ESP, stack-pointer value) before an interrupt occurs1." (IA-32 Intel® Architecture

Software Developer's Manual, Volume 2A: Instruction Set Reference, A-M, 253666-018, Jan 2006, "MOV—Move," page 3-584). And footnote 1. clearly states: "If a code instruction breakpoint (for debug) is placed on an instruction located immediately after a MOV SS instruction, the breakpoint may not be triggered." (page 3-585). For those who are new to how a debugger works, the "instruction breakpoint" which this refers to is not a breakpoint set by users, but rather the, let's call it, automatic breakpoint a debugger sets by itself on every single instruction users step into. So, according to these notes, what you may have thought was a "bug" in DEBUG, is in fact a processor doing what it was designed to do!

Does this mean we believe MS-DEBUG is completely "bug free"? No. In the future we'll post some examples here of real 'bugs' in DEBUG.

Before using any of the debugging (Trace, Procedure) or Register commands, you should familiarize yourself with the abbreviations for the CPU Registers that are referenced in DEBUG (See the Appendix, The 8086 CPU Registers for all the details.)

You should also know about the SEGMENT:OFFSET Addressing method used by DEBUG (and other programming utilities).

A note about where and how DEBUG is used in a computer's Memory:

Using DEBUG in a Windows® DOS-box (or CMD prompt) for the first time could easily confuse you! If you open two instances of DEBUG (one per DOS-window) and examine all the memory they can access, you might notice completely different data in many of the same memory locations! The reason is that each application under a Windows™ OS is (theoretically) given its own 4-Gigabyte "Virtual computer" sandbox to play in, and a copy of the critical data within the machine's first Megabyte of Memory is made for each running instance of DEBUG. Only under 16-bit DOS, does DEBUG actually have access to the real Memory locations in which the operating system itself is running; making it much easier to crash the whole system if an error is made. Under Windows®, the theory is that such errors should crash only the open CMD window or application that caused a problem, but not the whole computer! At least that's how Windows® is supposed to operate. From experience, it seems that Windows™ 2000/XP do a much better job at keeping control of their systems under the same circumstances that often ended in Blue Screens on the displays of earlier versions!

Quick Links to all the Commands

(Alphabetically Listed)

We recommend reading through the entire Tutorial on DEBUG before using these quick command links.

For a reminder of all the commands available while inside of DEBUG, simply enter a question mark (?) at the DEBUG prompt; when using DOS 5.0 or later.

Click on a command here for all its details:

```
-?
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]
```

Help on DEBUG Commands

For a reminder of all the commands (and most of the parameters) that are available while inside of DEBUG, simply enter a question mark (?) at the DEBUG prompt; when using DOS 5.0 or later. (Note: Expanded Memory commands are rarely if ever used and will not be discussed here.)

Quick Alphabetical Links

Click on a command here for all its details:

```
-?
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   (Learn 2's Complement!)
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 . . . . . . . . . (Learn this first!)
register    R [register]
search      S range list
trace/step  T [=address] [number]
unassemble  U [range]
write       W [address] [drive] [firstsector] [number]
```

How to Use the COMMANDS Parameters

NOTE: Parameters listed in brackets ([ ]) are optional. Optional parameters usually indicate there are a number of different ways a command could 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
```

For a detailed discussion, see: Segment:Offset notation.

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 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 all numbers and values used in any DEBUG command 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 when using DEBUG!

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 (which is true for the DEBUG version used by Windows™ XP) 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
-              -              -
```

Differences are always the second value subtracted from the first; AAA - 531 = 579. There are no carries past four digits.

Two's Complement arithmetic is always used in this calculator, so think of it as being limited to a maximum of plus 7FFFh (+ 32,767) or a minimum of minus 8000h (- 32,768). Positive values are represented by exactly the same digits as their numbers for 0000h through 7FFFh. A minus 7FFFh, however, is represented by the Hex digits 8001, and a minus 1h (-1) is represented by the Hex digits FFFF. Thus, the output of DEBUG after entering "h 4 fffc" would be a zero and an 8, because FFFC represents a minus 4h (-4) and 4 - (-4) = 8. Examples:

```
-h 4 fffc     -h 100 123     -h 7fff 8000
0000 0008     0223  FFDD     FFFF FFFF
-             -               -
```

Note the difference between 100h and 123h; what does FFDD represent? To find the numerical value of a Two's Complement number, first invert every bit (or find its logical inverse); that would be 0022, then add 1. So, this represents a negative 23h. Both the sum and the difference of 7FFFh and 8000h are a negative 1 (or FFFF); which can be arrived at using: 7FFFh + (- 8000h) = -1. However, it's much easier to think of the sums as having nothing to do with Two's Complement notation; thus, 7FFFh + 8000h = FFFFh (32,767 + 32,768 = 65,535). This will even hold true for the differences if the second value is less than the first. But any difference which produces a negative number, must be represented in Two's Complement.

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. The first example below shows what a Matrox video card on our system displayed.

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
-
```

```
-d 100 130
xxxx:0100 EB 24 0D 0A 54 68 69 73-20 69 73 20 6D 79 20 66  .$..This is my f
xxxx:0110 69 72 73 74 20 44 45 42-55 47 20 70 72 6F 67 72  irst DEBUG progr
xxxx:0120 61 6D 21 0D 0A 24 B4 09-BA 02 01 CD 21 B4 00 CD  am!..$......!...
xxxx:0130 21                                               !
-
```

The last example above, is what you'd see after entering the code under the Assemble command. We could just as easily have used the length option with the command 'd 100 l31' (that's an 'L' in front of the "31") to produce the same results above. The following example shows only the '$'-terminated display string, which has a length of 24h bytes (remember numbers in DEBUG are always hexadecimal); so that's 36 in decimal:

```
-d 102 124
xxxx:0100       0D 0A 54 68 69 73-20 69 73 20 6D 79 20 66    ..This is my f
xxxx:0110 69 72 73 74 20 44 45 42-55 47 20 70 72 6F 67 72  irst DEBUG progr
xxxx:0120 61 6D 21 0D 0A 24                                am!..$
-
```

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. [NOTE: In the examples below, if you do find the same data on your computer, the locations could easily vary from ours!]

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

-s 0:0 dff 'A20'
0000:0C42

-d 0:c40
0000:0C40 0D 0A 41 32 30 20 68 61-72 64 77 61 72 65 20 65  ..A20 hardware e
0000:0C50 72 72 6F 72 2E 20 20 43-6F 6E 74 61 63 74 20 74  rror.  Contact t
0000:0C60 65 63 68 6E 69 63 61 6C-20 73 75 70 70 6F 72 74  echnical support
0000:0C70 20 74 6F 20 69 64 65 6E-74 69 66 79 20 74 68 65   to identify the
0000:0C80 20 70 72 6F 62 6C 65 6D-2E 0D 0A 24 1A 00 BA F6   problem...$....

-s 0:0 dff 43 4f 4d
0000:0774
```

```
0000:07C2
0000:07D4
0000:07E6

-d 0:770
0000:0770 7A 02 A6 02 43 4F 4D 31-20 20 20 20 8E 00 70 00  z...COM1    ..p.
0000:0780 C0 A0 7A 02 91 02 4C 50-54 31 20 20 20 20 A0 00  ..z...LPT1    ..
0000:0790 70 00 C0 A0 7A 02 98 02-4C 50 54 32 20 20 20 20  p...z...LPT2
0000:07A0 2D 01 70 00 C0 A0 7A 02-9F 02 4C 50 54 33 20 20  -.p...z...LPT3
0000:07B0 20 20 11 EA 27 27 3F FD-CA 00 70 00 00 80 7A 02    ..''?...p...z.
0000:07C0 AC 02 43 4F 4D 32 20 20-20 20 DC 00 70 00 00 80  ..COM2    ..p...
0000:07D0 7A 02 B2 02 43 4F 4D 33-20 20 20 20 00 00 6B 03  z...COM3    ..k.
0000:07E0 00 80 7A 02 B8 02 43 4F-4D 34 20 20 20 20 E8 D2  ..z...COM4    ..
```

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 a whole segment 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

-f 100 ffff 0
```

This last example fills almost all of the assigned Segment with zero bytes (which can also be thought of as clearing the Segment). You should use this command whenever you want to be sure the bytes you'll be looking at in DEBUG's Segment are those you entered or loaded, or bytes DEBUG has changed; not soem previously used bytes from memory! If you want to examine a file from a disk in a 'clean' Segment, you'll first have to start DEBUG without any filename, clear the Segment using: f 100 ffff 0 and then finally load the file using the Name (n) and Load (L) commands in that order.

NOTE: Filling (clearing) any bytes in the area from 00h through FFh of our Segment can sometimes lead to problems; especially when file I/O is involved. DEBUG stores data for its own use in those locations, so we recommend you never overwrite bytes in that area; unless you have a reason for doing so!

Example: A student in an Assembly class was told to enter a string of commands under DEBUG, the last one being: JMP 0 which he was supposed to Trace (T) to the next command and then execute it. He was told it would be an INT 20 instruction. Well in most cases this is true, because DEBUG always sets the first two bytes of its working segment to "CD 20" for just this purpose. Let's test this out. First, open a new instance of DEBUG, then enter the following commands:

```
-f 100 ffff 0     [Zero-out 100 through FFFF]
-e 100 e9 fd fe   [Enters a 'JMP 0' at 100]
-u 100 102        [Check for correct entry]
xxxx:0100 E9FDFE        JMP     0000

-r
AX=0000   BX=0000   CX=0000   DX=0000   SP=FFEE   BP=0000   SI=0000   DI=0000
DS=xxxx   ES=xxxx   SS=xxxx   CS=xxxx   IP=0100    NV  UP  EI  PL  NZ  NA  PO  NC
xxxx:0100 E9FDFE        JMP     0000

-u 0 1
xxxx:0000 CD20          INT     20
```

If you don't see "INT 20" after entering "u 0 1", then restart DEBUG and try again.

`-t` [The "T"(Trace) command]

```
AX=0000  BX=0000  CX=0000  DX=0000  SP=FFEE  BP=0000  SI=0000  DI=0000
DS=xxxx  ES=xxxx  SS=xxxx  CS=xxxx  IP=0000   NV UP EI PL NZ NA PO NC
xxxx:0000 CD20          INT     20
```

`-p` [Always make sure you use a "P"(Proceed) command for Interrupts!]

Program terminated normally

`-q` [Quit]

Well, this never worked for those students. Why? Because the teacher had mistakenly told them to Fill the whole segment with zero bytes (f 0 ffff 0), in essence telling them to delete the very instruction he'd wanted them to execute!

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
```

This next example shows that either single(') or double(") quote marks are acceptable for entering ASCII data. By allowing both forms, entry strings can be created to include either type of quote mark as data:

```
-e 200 'An "ASCII-Z string" is always followed by '
-e 22a "a zero-byte ('00h')." 0
```

But in order to enter more than a single line of ASCII data, the A (Assemble) command is more practical since it will calculate the next offset for you! (See that command for a Memory dump of these bytes.) Now we'll examine a string of 11 hex bytes you can enter into Memory at locations CS:0100 and following:

```
-e 100 B4 09 BA 0B 01 CD 21 B4 00 CD 21
```

This is actually machine code for a program that will display whatever ASCII characters it finds at locations CS:010B and following, until it encounters a byte value of 24h (a $ sign). If you want to run this program, we'd recommend entering 24h at offset location 7EAh of the Segment so the program will terminate there:

```
-e 7ea 24
-g =100
```

And you'll soon see: "Program terminated normally" on the display screen. Why did we pick 7EAh? Because many DOS screens are set to display only 25 lines of 80 (50h) characters, and this value allows you to view the maximum number of characters possible on a single screen between the "Go," termination and prompt ("-") lines.

Here's something a bit more interesting for you to try out: It's essentially the same program, but the data includes all of the byte values from 00h through FFh; except for 24h which we placed at the end of the last line. The DEBUG prompt symbol, - , has been purposely excluded from the lines below, so you can copy and paste the whole block into a DEBUG DOS-box (Help on using DOS-Window controls is here if needed):

```
e 100 B4 09 BA 0B 01 CD 21 B4 00 CD 21 0D 0A 0D 0A 00 01 02
e 112 03 04 05 06 07 08 09 20 0B 0C 20 0E 0F 10 11 12 13 14
e 124 15 16 17 18 19 1A 1B 1C 1D 1E 1F 20 21 22 23 20 25 26
e 136 27 28 29 2A 2B 2C 2D 2E 2F 30 31 32 33 34 35 36 37 38
e 148 39 3A 3B 3C 3D 3E 3F 0D 0A 0D 0A 40 41 42 43 44 45 46
e 15a 47 48 49 4A 4B 4C 4D 4E 4F 50 51 52 53 54 55 56 57 58
e 16c 59 5A 5B 5C 5D 5E 5F 60 61 62 63 64 65 66 67 68 69 6A
e 17e 6B 6C 6D 6E 6F 70 71 72 73 74 75 76 77 78 79 7A 7B 7C
e 190 7D 7E 7F 0D 0A 0D 0A 80 81 82 83 84 85 86 87 88 89 8A
e 1a2 8B 8C 8D 8E 8F 90 91 92 93 94 95 96 97 98 99 9A 9B 9C
e 1b4 9D 9E 9F a0 a1 a2 a3 a4 a5 a6 a7 a8 a9 aA aB aC aD aE
e 1c6 aF b0 b1 b2 b3 b4 b5 b6 b7 b8 b9 bA bB bC bD bE bF 0D
e 1d8 0A 0D 0A c0 c1 c2 c3 c4 c5 c6 c7 c8 c9 cA cB cC cD cE
e 1ea cF d0 d1 d2 d3 d4 d5 d6 d7 d8 d9 dA dB dC dD dE dF e0
e 1fc e1 e2 e3 e4 e5 e6 e7 e8 e9 eA eB eC eD eE eF f0 f1 f2
e 20e f3 f4 f5 f6 f7 f8 f9 fA fB fC fD fE fF 0D 0A 0D 0A 24
```

The bytes 0Dh and 0Ah produce a Carriage Return and Linefeed on the display, so we replaced them in the listing above by 20h; a SPACE byte. The 24h byte was moved to the end of the program with another 20h taking its place. The bytes shown above in blue (0D 0A 0D 0A) form blank lines at the beginning of the output and after every 64 bytes for a nice formatted display.

Therefore, when the program is run, we should see four separate lines of 64 characters each (a few of those being blank spaces as mentioned above), right? Well, let's find out: Start DEBUG in a DOS-Window, copy and paste the lines above into DEBUG at its prompt symbol, then enter the following command:

```
g =100    ( 'g' followed by a SPACE, then '=100')
```

This will immediately run (see Go command) the program, displaying the output lines followed by: "Program terminated normally" [Do not exit DEBUG, just leave the window open. We're going to show you how to 'patch' this code and save the results as a nice little console program].

Were you surprised to find more than four spaces on the first line; starting with the very first byte? What about the fact we appear to have missing characters at the end of that line? We'll briefly explain why the characters appeared this way on your screen, but in order to create programs of your own, you'll need to study about control characters, Interrupts and what effect different BIOS and DOS video functions have on the way ASCII characters are displayed. OK, here's what happened:

First, the Zero byte also displays as a blank space here. The 07 byte may make a beep or ding sound (but does not display anything), 08 performs a BACKSPACE (erasing the 06 byte character) and 09 is a TAB -- which may jump up to eight columns to the right before reaching the next 'Tab Stop.' But since it just happens to begin in column seven, it only moves one column to the right where our program places the space we substituted for 0Ah. Lastly, for some reason, when using Function 09 of INT 21h ("Display a string of characters until a '$' sign is encountered"), the ESC character (1Bh; 27 decimal) doesn't display or do anything. So, after reaching the end of the first line, it only appeared as if many of the characters we expected to see were never displayed. In reality, the last three characters are there. It's because of the bytes 07h (displayed nothing), 08h (only backspaced over 06h), 09h (displayed nothing, but moved cursor forward one byte) and 1Bh (displayed nothing) that we saw what we did.

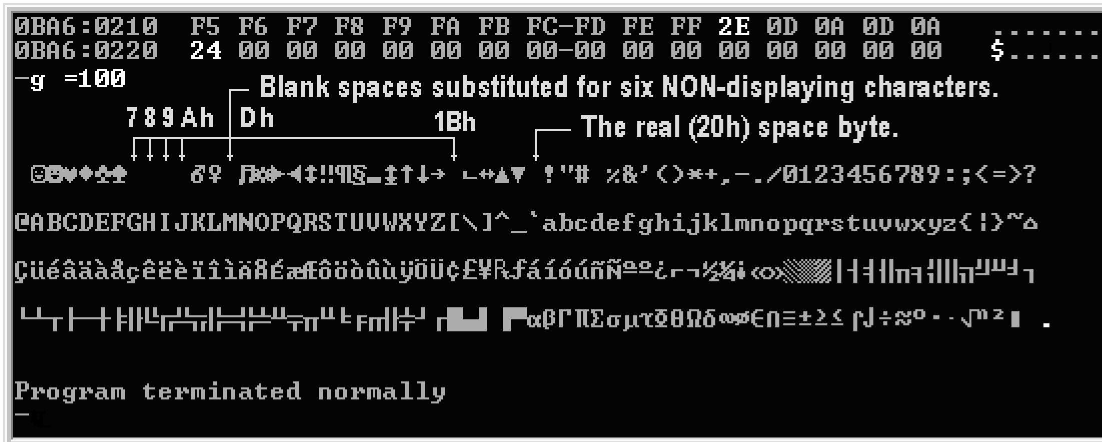
Enter the following two lines into DEBUG (which contain more blank-space substitutions), run the program again, and you'll see all the displayable characters output on the first line in their correct positions:

```
e 10F 00 01 02 03 04 05 06 20 20 20 20 0B 0C 20
e 11D 0E 0F 10 11 12 13 14 15 16 17 18 19 1A 20
```

All four rows will display evenly in size, including the last one. But the last character, FFh (255 decimal), just like the first, also displays as a blank space here! You can prove this by inserting another byte such as 2Eh (a period '.') after FFh. We've created the following patch which effectively moves up the remainder of the program after the FFh by one:

```
e 21b 2e 0d 0a 0d 0a 24
```

After patching and running it again, the program output should look like this:



If you want to, you can save this code as an executable program by first giving it a path and filename (such as, C:\TEMP\ASCIIDSP.COM; see Name command) and writing the bytes (see Write command) to a file like this:

```
-n c:\temp\asciidsp.com
-rcx
CX0000
:121        [ Program Length = 220h - 100h + 1 = 121h ]
-w
```

If you check the file properties of ASCIIDSP.COM, its size should be 289 bytes.

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 where to start executing code. 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 any instruction which requires a CPU above an 8088/8086)!

Using "Go" after setting breakpoints at instructions you're sure DEBUG understands is one way you can get more use out of this program; for example, when debugging the "real" code of a Master Boot Record written for 80386 or above processors. DEBUG won't be able to disassemble nor single step through such code, but it can still pass the instructions to the CPU for execution then stop for breakpoints you've set at any instruction it does understand.

CAUTION: DEBUG replaces the original instructions of any listed breakpoint addresses with CCh (an INT 3). The instructions at these locations are restored to their original bytes 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 unless you're sure you've cleared all your breakpoints! (Saving to a backup copy before ever using a breakpoint is often the best 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); spaces after the commas would make it clearer but aren't necessary.

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.

Examples:

Using the character string from our E (Enter) command above:

```
-a 200
xxxx:0200 db 'An "ASCII-Z string" is always followed by '
xxxx:022A db "a zero-byte ('00h').", 0
xxxx:023F

-d 200 23e
xxxx:0200  41 6E 20 22 41 53 43 49-49 2D 5A 20 73 74 72 69   An "ASCII-Z stri
xxxx:0210  6E 67 22 20 69 73 20 61-6C 77 61 79 73 20 66 6F   ng" is always fo
xxxx:0220  6C 6C 6F 77 65 64 20 62-79 20 61 20 7A 65 72 6F   llowed by a zero
xxxx:0230  2D 62 79 74 65 20 28 27-30 30 68 27 29 2E 00      -byte ('00h')..
```

ENTER the characters in bold type; you do not need to enter the comments after the semi-colon (;) symbols:

```
-a 100
xxxx:0100 jmp 126       ; Jump over data that follows:
xxxx:0102 db 0d,0a,"This is my first DEBUG program!"
xxxx:0123 db 0d,0a,"$"
xxxx:0126 mov ah,9      ; Function 09 of Int 21h:
xxxx:0128 mov dx,102    ; DS:DX -> $-terminated string.
xxxx:012B int 21        ; Write String to STD Output.
xxxx:012D mov ah,0      ; Function 00 of Int 21h:
xxxx:012F int 21        ; Terminate Program.
xxxx:0131
-g =100

This is my first DEBUG program!

Program terminated normally
-
```

NOTE: You can pipe simple 8086/8088 Assembly Language "scripts" into DEBUG (You can even include a semi-colon ';' followed by comments on most of its lines. For some odd reason though, these comments do not appear to be allowed on DB/DW lines!). For example, you can copy and paste the following into DEBUG (after entering an initial "a" command) and obtain the same results as above:

```
jmp 126       ; Jump over data that follows:
db  0d,0a,"This is my first DEBUG program!"
db  0d,0a,"$"
; End of string marker above: "$"=24h
mov ah,9      ; Function 09 of Int 21h:
mov dx,102    ; DS:DX -> $-terminated string.
int 21        ; Write String to STD Output.
mov ah,0      ; Function 00 of Int 21h:
int 21        ; Terminate Program.
```

DEBUG uses the convention of enclosing operands which refer to Memory locations in square brackets '[ ]' (as opposed to an immediate value as an operand).

DEBUG may require you to explicitly tell it whether or not an operand refers to a word or byte in Memory! In such cases, the data type must be stated using the prefixes 'WORD PTR' or 'BYTE PTR' For all 8087 opcodes, the WAIT or FWAIT prefix must be explicitly specified.

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 newer x86 instructions (those for the 80286, 80386 on up to the lastest CPU from Intel; which are all beyond the ability of DEBUG to understand)!

Example:

```
-u 126 12F
xxxx:0126 B409          MOV     AH,09
xxxx:0128 BA0201        MOV     DX,0102
xxxx:012B CD21          INT     21
xxxx:012D B400          MOV     AH,00
xxxx:012F 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
```

The first space isn't necessary under most versions of DEBUG; so you can try to get away with just "o70" and "i71" instead. Here's a page of more complex examples dealing with hard drives and the ATA commands for reading info directly from a disk controller!

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).

The terms 'Volume' or 'Logical Drive' used in the definition above mean you cannot use the 'L' command to load or examine the MBR, or any other sectors outside the Primary Volumes or Logical Drive Letters assigned by DOS or Windows! For example (under Windows™ 9x/ME), if you enter the command: L 100 2 0 1 in DEBUG, instead of seeing the very first sector on that hard disk (the MBR), you'll see the first sector of the Boot Record for the Logical drive C: instead (the first partition that can accessed by a compatible MS-DOS or Windows OS). This and the following comments about diskettes, show that DEBUG has always been quite limited compared to a good disk editor or the UNIX 'dd' program.

Load can still be useful in examining Floppy Disks even under Windows™ 2000/XP, but (unfortunately), only if the disk can be read by MS-DOS or Windows. Once again, this shows how limited DEBUG is compared to any utility that can view the raw data on either a hard drive or diskette. (For those of you who wish to examine the actual contents of a hard disk under Windows™ XP, there are free disk editors, such as HxD, which allow you to do so.)

Unlike hard disks, the very first sector on a floppy disk is an OS Boot sector. Here's what you might see from a Logical disk sector and some dumps from a couple floppy disks.

Examples:

```
-l 100 2 0 1   [ the C: drive. ]
-d 100 10f
xxxx:0100  EB 58 90 4D 53 57 49 4E-34 2E 31 00 02 08 20 00  .X.MSWIN4.1... .
-d 280 2ff
xxxx:0280  01 27 0D 0A 49 6E 76 61-6C 69 64 20 73 79 73 74  .'..Invalid syst
xxxx:0290  65 6D 20 64 69 73 6B FF-0D 0A 44 69 73 6B 20 49  em disk...Disk I
xxxx:02A0  2F 4F 20 65 72 72 6F 72-FF 0D 0A 52 65 70 6C 61  /O error...Repla
xxxx:02B0  63 65 20 74 68 65 20 64-69 73 6B 2C 20 61 6E 64  ce the disk, and
xxxx:02C0  20 74 68 65 6E 20 70 72-65 73 73 20 61 6E 79 20   then press any
xxxx:02D0  6B 65 79 0D 0A 00 00 00-49 4F 20 20 20 20 20 20  key.....IO
xxxx:02E0  53 59 53 4D 53 44 4F 53-20 20 20 53 59 53 7E 01  SYSMSDOS   SYS~.
xxxx:02F0  00 57 49 4E 42 4F 4F 54-20 53 59 53 00 00 55 AA  .WINBOOT SYS..U.
-

-l 100 0 0 1   [ a floppy in the A: drive. ]
-d 100 13d
xxxx:0100  EB 3C 90 29 47 38 71 33-49 48 43 00 02 01 01 00  .<.)G8q3IHC.....
xxxx:0110  02 E0 00 40 0B F0 09 00-12 00 02 00 00 00 00 00  ...@............
xxxx:0120  00 00 00 00 00 00 29 40-16 D8 13 4E 4F 20 4E 41  ......)@...NO NA
xxxx:0130  4D 45 20 20 20 20 46 41-54 31 32 20 20 20        ME    FAT12
-

-l 100 0 0 1   [ a different floppy in the A: drive. ]
-d 100 13d
xxxx:0100  EB 3C 90 53 59 53 4C 49-4E 55 58 00 02 01 01 00  .<.SYSLINUX.....
xxxx:0110  02 E0 00 40 0B F0 09 00-12 00 02 00 00 00 00 00  ...@............
xxxx:0120  00 00 00 00 00 00 29 7E-CF 55 3C 20 20 20 20 20  ......)~.U<
xxxx:0130  20 20 20 20 20 20 46 41-54 31 32 20 20 20              FAT12
-

-d 2d0 2ff
xxxx:02D0  42 3B 16 1A 7C 72 03 40-31 D2 29 F1 EB A7 42 6F  B;..|r.@1.)...Bo
xxxx:02E0  6F 74 20 66 61 69 6C 65-64 0D 0A 00 00 00 00 4C  ot failed......L
xxxx:02F0  44 4C 49 4E 55 58 20 53-59 53 F4 3C 82 3A 55 AA  DLINUX SYS.<.:U.
```

The Linux Boot disk above (note the word: SYSLINUX) is the kind formatted as an MS-DOS diskette and not with a true Linux file system (such as ext2 or ext3). If it had been formatted with some other kind of file system, or had a faulty boot sector, then MS-DEBUG would not be able to read it! Instead you'd see that old "General failure reading drive A / Abort, Retry, Fail?" error message! And when you had finally cleared away that error message, you'd be greeted by DEBUG's "Disk error reading drive A" error message. This makes DEBUG almost worthless as far as trying to fix an error in a floppy disk's boot sector! However, if you keep a binary copy of a good floppy disk Boot Sector somewhere, you could use DEBUG to overwrite whatever's on a faulty floppy disk's first sector (see Write command). But if you really need to see what's in such a Boot sector (i.e., what is keeping DEBUG from recognizing it as valid), you'll need to use a disk editor such as Symantec's Norton DiskEdit (in Physical disk Mode only).

NOTE: Just because a floppy disk can't be read by DOS or opened in DEBUG does NOT necessarily mean it's defective. It might simply have been formatted with a file system it cannot recognize (such as Linux's ext2) and could easily boot-up on its own; a very good reason for labeling your disks!

(CAUTION: Never try booting your system with a disk you're not 100% sure of; unless you disconnect all hard disks and don't have any flash BIOS, since it might contain a nasty boot virus!)

[Many floppy disks have the letters IHC in their OEM ID field. What kind of OEM Name is that? None. Someone at Microsoft decided that this was where they'd place a new pseudo-random type of identification to make sure that any information cached by 'Windows 9x' from one disk wouldn't be mixed up with info from a different one if you swapped disks. The whole string begins with five pseudo-random hex bytes, and always ends with the characters IHC. All floppy diskettes that are not write-protected will have any original OEM ID overwritten. Once Windows has written this string, it will remain the same for any future disk reads or writes. However, performing even a quick format under Windows, will change the five hex bytes every time.

Some have concluded the characters 'IHC' are the first three letters of the word "Chicago" in reverse order, since Chicago was the 'code name' for Windows™ 95 before it was ever released (it would have appeared as ' OGACIHC' on the hypothetical disk). Although certainly a possibility, I have no proof of that. Due to our interest in some very old Greek Manuscripts, we still can't help but see the 3 characters 'IHC' as an Iota, Eta and old style Sigma since this combination of letters was often used as an abbreviation for the Greek word "IHSUS" (Jesus). Just another of many coincidences in our lives.

REMEMBER: If you really want to preserve all of the contents of an important diskette, you can't even perform a simple Directory read under a Windows OS, UNLESS it is 'write-protected' and you know the drive's write-protect system is functioning correctly!]

Move: M range address

This command could be called COPY (not Move), since it only copies all the bytes from within the specified range to a new address.

Examples:

```
-m 7c00 7dff 600
```

Copies all 512 (200h) of the bytes between Offsets 7C00 and 7DFF (inclusive) to Offset 0600 and following.

```
-m 100 2ff 200
```

This second example shows it's very easy to overwrite much of the same source area you're copying from when using the Move command. However, DEBUG must store all the source bytes in Memory before writing them; otherwise, this example would cause a problem when overwriting an area it hadn't copied data from yet, if it were copying only one byte at a time from that source area! The example above copies all 512 bytes of offsets 100h through 2FFh (inclusive) to Offsets 0200h and following; overwriting the last 256 (2FF-200+1 hex) bytes of the source in the process. This is also true under real 16-bit DOS.

Note: If your Move command produces a situation where offset FFFFh has already been written to yet there's still more data to write, you may experience unexpected results! Remember, DEBUG is technically assigned to only one 64 KiB Segment. So, the data will wrap around to the beginning of the Segment, possibly overwriting some of the source bytes you told it to copy from! But other symptoms may occur as well, since the first area to be overwritten after wrapping around (00h through

FFh) is sometimes used by DEBUG to keep track of itself. So do the math whenever copying bytes to a higher location in Memory, to be sure you don't run out of room at the end of the Segment.

Copying bytes to a lower location in the Segment is quite simple though; DEBUG could actually copy just one byte at a time in that direction and never overwrite a source byte before it was already copied.

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)!

Note: Unlike the other programs listed on this page, this one uses Function 4Ch instead of Function 00 of Interrupt 21h to terminate its execution. This is the preferred termination function for most DOS programs, because it can not only send a "Return Code" (an ERRORLEVEL value; of whatever is in the AL register), but will also close all open files and free all memory belonging to the process. When you use this function to terminate a program running under DEBUG though, it has a tendency to also terminate DEBUG itself; thus our reason for rarely using it here!

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
```

For an explanation of the names of the registers (AX, BX, CX, etc. and the Flag symbols: NV UP EI PL NZ NA PO NC), see the Appendix (The 8086 CPU Registers). The last line shows the next CPU instruction (actually the first in this case) to be executed, begins at memory location 100 hex (the offset) in Segment ED8 hex (0ED8:0100) and the Hex bytes E90E01 represent the actual binary machine code of the CPU instruction (JMP 0211 in Assembly language) that would be executed by DEBUG if you entered a Trace (t) or Proceed (p) command.

If you enter the ' r ' followed by the abbreviation for an 8086 register, such as: ' rcx ', then DEBUG will display only the contents of that register followed by a line with a colon symbol (:) on which you can enter a hex number to change the contents of that register. If you simply press the ENTER key, the contents remain the same. For example:

```
-rcx
CX 0100
:273
```

means the Register command was used to change the contents of the CX register from 0100 to 0273. The command rcx could be used again to verify the change had indeed taken place. If you type the letter f after an r: rf, this commands DEBUG to display all the FLAG register bits with a prompt on the same line which allows you to change any or none of the individual flag bits. For example, here's how you would display the flags and change just the Zero Flag bit from being cleared (a 0 bit) to being set (a 1 bit):

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

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

As you can see above the Zero Flag was changed from NZ (cleared) to ZR (set). See our Appendix: The FLAGS Register for an explanation of all the Flag abbreviations.

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 by itself, it will normally 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; or step into BIOS code which has been shadowed in read-only portions of Memory for decades now.) NOTE: If you find yourself stuck inside a long LOOP or REPeat string instruction, enter a P (Proceed) command and it will complete the operation and move to the next instruction.

Proceed: P [=address] [number]
Proceed acts exactly the same as Debug's T (Trace) command for most instruction types; with these notable exceptions: Proceed will immediately execute all instructions (rather than stepping through each one) inside any subroutine CALL, LOOP, REPeat string instruction or any software INTerrupt. You can still step into an INT or execute all the code contained in a subroutine if you need to, but with the Proceed (P) command you are not required to do so.

This means Proceed will probably be the command of choice for most of debugging tasks, with Trace only being used to step through an unfamiliar subroutine or check the logic of the first few iterations of long REP string instructions or LOOPs. And it's a must use command when it comes to Interrupts!

Write:
W [address] [drive] [firstsector] [number]

WARNING 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 non-booting system! (Although Windows XP and later prevent direct Sector writes to a hard disk, they are still allowed to floppy drive media, i.e., drive letters A: or B:)

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 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 in the following example from our MyMBR Batch/Debug Script Program.

EXAMPLE:

After creating and running a small program inside of DEBUG which copies the Master Boot Record (MBR) to Offset 0000h through 01FFh, these DEBUG commands save the MBR to a file on the hard disk:

```
-n mymbr.bin
-rcx
CX 0001
:200
-w 0
Writing 00200 bytes   [ 512 bytes in decimal ]
-
```

The BX register had already been set to zero by a previous instruction, so the CX register was simply set to 200 and the WRITE command executed with an address of 0 (if no address is used, the Write command starts saving bytes at Offset 100).

The WRITE command can, however, be used in a relatively safe manner with Floppy disks. For example, you could use the Load (L) command:

```
l 7c00 0 0 1
```

to load the first sector of an MS-DOS or Windows floppy disk into DEBUG's memory at location 7C00, change some of the code and/or messages (if you know how to do so) and then use the 'W' command:

```
w 7c00 0 0 1
```

to write the changes back to the floppy disk's first sector.

---

*Although the BX and CX registers are often referenced in books on Assembly as BX:CX when they discuss this write command, note that these registers are not being used like Segment:Offset pairs in this case! They are a true combination of higher and lower bytes which form a 'double word' for a theoretical total of about 4 GB (FFFF FFFFh = 4,294,967,295 bytes) that could be written to a file! Whether or not this is true of all versions of DEBUG, under DOS 7.1, we've been able to load image files of several hundred KiB and write the whole file to a new location!

For example, if you load a 360 KiB image file into DEBUG at a DOS prompt, then check the registers, BX will equal 0005 and CX will contain A000. The major problem here though is the fact DEBUG uses CONVENTIONAL MEMORY, so trying to load a file greater than about 400 KiB is bound to elicit an "Insufficient Memory" error!