EEM 332 - Microprocessors
Introduction to DEBUG

Notes are taken from;
*DEBUG/ASSEMBLY TUTORIAL by Fran Golden
http://www.datainstitute.com/debug1.htm
**A Guide to DEBUG by Daniel B. Sedory
http://www.geocities.com/thestarman3/asm/debug/debug.htm

COM Extensions

1. COM programs are very small and compact programs that cannot be larger than 65K bytes in size.
2. The PSP of a COM program is located in the first 100 Hex (256 Dec) locations of the program.
3. The first instruction of the COM program must start at offset 100 in memory.
4. DOS creates the PSP for the COM program, which means we don't have to be concerned with this when we assemble a program.
5. All the data, code, and the stack area are in the same segment of memory. (1 segment is 65K)

EXE Extensions

1. The EXE programs can be any size from (200 bytes-640k bytes)
2. The PSP must be setup by the programmer, when the program is assembled.
3. The programmer determines where the first instruction is in the program.
4. The EXE program uses separate segments for the data, code and stack area in memory.
From the above comparison you can see it is much more difficult to assemble an EXE program than it is a COM program. The debug utility program was designed to work only with a COM program by setting up the PSP area each time we enter debug. Once in debug, we can start assembly of a program at offset 100 and not be concerned with PSP or where the data, code, and stack is located. It is possible to look at an EXE program with debug if we rename the program with a different extension before we load it into memory.

Assembly language

Programming in the early days of computing had to be written using the machine code of the microprocessor and as a result required specialized people who had this ability. The use of computers was restricted to the companies and corporations with the finances to support such a specialized operation. It didn't take long for the programmers to write the code to do most of the hard work in binary. One of the most important programs written was the Assembler, which was a conversion type program that allowed the use of English words to represent the instruction set of the CPU, that was compiled into machine code after they were written. One such program is Debug.com that is supplied by the MicroSoft corp. in most versions of DOS.

Debug, which is a small but powerful assembler, converts English sounding words called mnemonics into the machine code. All instructions for the 8088 have a mnemonic word to represent them in debug. For example, in the program above, the code can be written in an assembler language that is much easier to write and understand.

The mnemonic form of that program is shown below.

mnemonic form---------machine code

MOV AX,0005------------B80500
ADD BX,AX---------------01C3
MOV [120],AX------------891E2001

Details of Each Command

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
-
```

NOTE: You can *pipe* simple 8086/8088 Assembly Language scripts into DEBUG (which can even include a semi-colon ';' followed by comments on any of its lines!).

- 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.

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 on either side of them).

Dump: D [range]
D [address] [length]

Displays the contents of a block of memory.

Example:

```
-d 100 133
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 31 C0-B4 09 BA 02 01 CD 21 B8  am!..$1.......!.
xxxx:0130 4C 00 CD 21                                      L..!
-
```

This next example uses the option [**length**]; the '**l24**' means a length of 24 hexadecimal bytes (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!..$
-
```

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

Now let's examine a string of 14 hex bytes entered into Memory at location CS:0100 and following:

-e 100 31 C0 B4 09 BA 0E 01 CD 21 B8 4C 00 CD 21

This is actually machine code for a program that will display whatever it finds at location CS:010E until it encounters a byte of 24h (a '$' sign).

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 bytes from 00h through FFh; except for 24h which was placed at the end of the last line. The DEBUG prompt '-' is purposely left out here so you can copy and paste the whole block all at once into DEBUG using a DOS-Window (You'll find Help on using DOS-Window controls here):

```
e 100 31 C0 B4 09 BA 0E 01 CD 21 B8 4C 00 CD 21 0D 0A 0D 0A
e 112 00 01 02 03 04 05 06 07 08 09 20 0B 0C 20 0E 0F 10 11
e 124 12 13 14 15 16 17 18 19 1A 1B 1C 1D 1E 1F 20 21 22 23
e 136 20 25 26 27 28 29 2A 2B 2C 2D 2E 2F 30 31 32 33 34 35
e 148 36 37 38 39 3A 3B 3C 3D 3E 3F 0D 0A 0D 0A 40 41 42 43
e 15a 44 45 46 47 48 49 4A 4B 4C 4D 4E 4F 50 51 52 53 54 55
e 16c 56 57 58 59 5A 5B 5C 5D 5E 5F 60 61 62 63 64 65 66 67
e 17e 68 69 6A 6B 6C 6D 6E 6F 70 71 72 73 74 75 76 77 78 79
e 190 7A 7B 7C 7D 7E 7F 0D 0A 0D 0A 80 81 82 83 84 85 86 87
e 1a2 88 89 8A 8B 8C 8D 8E 8F 90 91 92 93 94 95 96 97 98 99
e 1b4 9A 9B 9C 9D 9E 9F a0 a1 a2 a3 a4 a5 a6 a7 a8 a9 aA aB
e 1c6 aC aD aE aF b0 b1 b2 b3 b4 b5 b6 b7 b8 b9 bA bB bC bD
e 1d8 bE bF 0D 0A 0D 0A c0 c1 c2 c3 c4 c5 c6 c7 c8 c9 cA cB
e 1ea cC cD cE cF d0 d1 d2 d3 d4 d5 d6 d7 d8 d9 dA dB dC dD
e 1fc dE dF e0 e1 e2 e3 e4 e5 e6 e7 e8 e9 eA eB eC eD eE eF
e 20e f0 f1 f2 f3 f4 f5 f6 f7 f8 f9 fA fB fC fD fE fF 0D 0A
e 220 0D 0A 24
```

The bytes **0D**h and **0A**h produce a Carriage Return and Linefeed on the display, so they were replaced by a **20**h (a space) in the listing above. The **24**h byte was, of course, moved to the end of the listing and its original location also turned into a space. A blank line was placed at the beginning and after every 64 bytes in the listing using the sequence '0D 0A 0D 0A.' So, we should see four separate lines of 64 characters each (a few of those being blank spaces in the first line of course) when the program is run, right? Start DEBUG in a DOS-Window, copy and paste the lines above into DEBUG at the '-' prompt, then enter the following command:

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

This will run***** the program in DEBUG... displaying the lines we mentioned and another stating: "Program terminated normally." [If possible, do NOT close the DOS-Window or exit DEBUG as we'll be making a 'patch' to the code later.] If you were surprised to find more than four spaces on the first line, or that there appear to be some missing characters at the end of that line, then you might want to study about the Control Characters at the beginning of an ASCII chart. You'll also need to learn about Interrupts and what effect different BIOS and DOS Video Functions have on how the ASCII Characters are displayed. (*****Refer to: The 'G' Gocommand.)

OK, I'll save you the time and tell you what happened: First, the Zero byte displays as a blank space here. The **07**h byte sounds a BEEP (but does not display anything), **08**h performs a BACKSPACE (erasing the 06h byte character) and **09**h 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 blank space we substituted for **0A**h. Lastly, for some reason, when using Function 09 of INT 21 ("Display a string of characters until a '$' sign is encountered"), the ESC character (**1B**h;

27 decimal) doesn't display or do anything either. So, upon reaching the end of the first line, it appears as if five of the characters were never displayed.

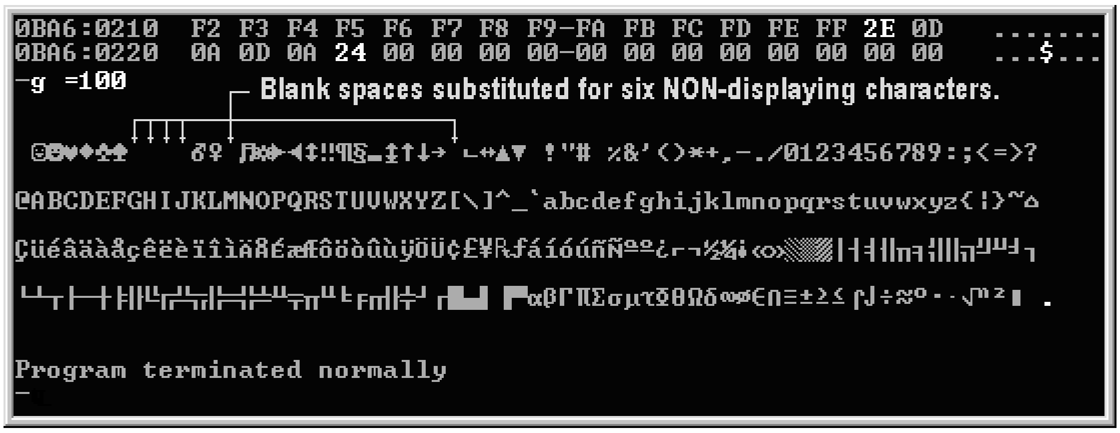
If you enter the following two lines into DEBUG (with more blank-space substitutions inserted) and run the program again, you'll see all of the displayable characters output on the first line in their correct positions:

```
e 112 00 01 02 03 04 05 06 20 20 20 20 0B 0C 20
e 120 0E 0F 10 11 12 13 14 15 16 17 18 19 1A 20
```

All four rows should come out even now, including the last one, because the **FF**h (255 decimal) displays as a blank space! You can prove this by inserting another byte such as 2Eh (a period '.') right after the FFh byte. Since DEBUG has **no** 'Insert' command, here's another 'patch' for you which effectively *moves* the remaining program bytes up one location:

e 21e 2e 0d 0a 0d 0a 24

Here's how the program output should now look when run again:



```
0BA6:0210  F2 F3 F4 F5 F6 F7 F8 F9-FA FB FC FD FE FF 2E 0D   ........
0BA6:0220  0A 0D 0A 24 00 00 00 00-00 00 00 00 00 00 00 00   ...$....
-g =100
 ☺☻♥♦♣♠    ♂♀ ♪♫☼►◄↕‼¶§▬↨↑↓→ ∟↔▲▼ !"# %&'()*+,-./0123456789:;<=>?
@ABCDEFGHIJKLMNOPQRSTUVWXYZ[\]^_`abcdefghijklmnopqrstuvwxyz{¦}~⌂
ÇüéâäàåçêëèïîìÄÅÉæÆôöòûùÿÖÜ¢£¥₧ƒáíóúñÑªº¿⌐¬½¼¡«»░▒▓│┤╡╢╖╕╣║╗╝╜╛┐
└┴┬├─┼╞╟╚╔╩╦╠═╬╧╨╤╥╙╘╒╓╫╪┘┌█▄▌▐▀αßΓπΣσµτΦΘΩδ∞φε∩≡±≥≤⌠⌡÷≈°∙·√ⁿ²■ .

Program terminated normally
-
```

You can save this program to your hard drive, by first giving it a path and filename (such as, C:\TEMP\ASCIIDSP.COM; see 'N' Name command) and writing the bytes (see 'W' Write command) to a file like this:

```
-n c:\temp\asciidsp.com
-rcx
CX0000
:124          [ Program Length = 223h - 100h + 1 = 124h ]
-w
```

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
```

```
-f 0 ffff 0
```

This last example fills *all* of the assigned Segment with bytes of zeros (which could also be thought of as *clearing* the whole Segment). I use this command whenever I want to be sure that the bytes I'm looking at are those I'm concerned about and not just random bits of code lost in memory!

If you want to examine a file from disk in a 'clean' Segment, then you would first have to start DEBUG without the filename on the command line, *clear the Segment* using -f 0 ffff 0, and then finally load the file with the Name (n) and Load (L) commands.

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.)

Hex: H value1 value2

A very simple (add and subtract only) hex calculator. Enter two Hex values (only up to four hex digits each), and DEBUG shows first the SUM and then the DIFFERENCE (no *carries* past the fourth digit). Examples:

```
-h 123 100          -h 100 123          -h ffff 1
0223  0023          0223  FFDD          0000  FFFE
-                   -                   -
```

Input: I port

The use of I/O commands while running Windows may be unreliable! This is especially true when trying to directly access hard disks through I/O commands! Long ago (when DOS was the only OS for PCs), there were dozens of **BASIC** programs that used I/O commands for all sorts of tasks through the parallel and serial ports (e.g., to change fonts on a printer or values in a modem's control registers). They can also be used for direct communications with your keyboard or floppy drive's control chips.

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

```
-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 wasn't necessary under my version of DEBUG; so you might be able to get away with "o70" and "i71" instead. Here's a page of more complex examples dealing with hard drives and the ATA commands for reading info from the disk controllers!

Load:

L [address] [drive] [firstsector] [number]

This command will LOAD the selected number of sectors from any disk's Logical Drive under the control of MS-DOS/Windows into Memory. The **address** is a location in Memory (use only 4 hex digits to keep it within the memory allocated to DEBUG) the data will be copied to, the **drive** number is mapped as: 0=A:, 1=B:, 2=C:, etc., **firstsector** counts from ZERO to the largest sector in the disk volume and **number** specifies **in hexadecimal** the total number of sectors that will be copied into Memory.

The terminology 'Disk Volume' or 'Logical Drive' means that you can not use L[oad] to load or examine the MBR, for example, or any other sectors outside of the Logical Drive Letters or Volumes assigned by DOS/Windows.
For example, if you enter the command: **L 100 2 0 1** in DEBUG, instead of seeing the very first sector on a hard disk (the MBR), you'll see the first sector of the Boot Record for the Logical drive C: of the first hard disk partition that can accessed by a compatible MS-DOS or Windows OS.

Load can be useful in examining a Floppy Disk, ***but, unfortunately, only if the disk can be read by MS-DOS/Windows.*** 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 ***d****umps* 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.
```
Note that the Linux Boot disk above is the kind formatted as an MS-DOS diskette and not as a true Linux floppy. If it had been a floppy formatted for some other OS or with 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 then DEBUG's "Disk error reading drive A" message after having cleared the first 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 want to see what's in such a Boot sector (that keeps DEBUG from recognizing it as valid), you'll need to use a disk editor that will allow you to do so; such as Symantec's Norton DiskEdit (ONLY in Physical Disk Mode though).

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 different OS (such as Linux) 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, because 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 that 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 my interest in some very old Greek Manuscripts, I still can't help seeing 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. Just another one of those coincidences of life.

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 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 at all) 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).

The 'N' command makes it quite easy to save data or an Assembly program you created in DEBUG to a file on your hard drive!

For example, these commands (in BOLD; with the DEBUG reponses):

```
-n c:\temp\dosok.com
-a 100
cs:0100 jmp 12C
cs:0102 db "It's OK to run this program under DOS!"
cs:0128 db 0d,0a,24
cs:012B nop
cs:012C mov dx,102
cs:012F mov ah,9
cs:0131 int 21
cs:0133 mov ax,4c01
cs:0136 int 21
cs:0138
-rcx
CX 0000
:38
-w
Writing 00038 bytes     [ 56 bytes in decimal ]
-q
```

will create a 56-byte file called DOSOK.COM in the C:\TEMP folder. Even when running DEBUG in a DOS-window, though, the file names are still limited to DOS's 8 characters plus 3 for the extension!

Output: O port byte

See the comments under the **Input** command.

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.

Quit: **Q**

Immediately quits (exits) the Debug program! No questions ever asked... should be the easiest command to remember.

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 that 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 that 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 that 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 that the change had indeed taken place. If you type the letter **f** after an r: ' rf ', this commands DEBUG to display all of 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 the Appendix The FLAGS Register below for an explanation of all the Flag abbreviations.

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

-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...$....

-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    ..
```

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.)

***Un*assemble: 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 lastest 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
-
```

The 8086 CPU REGISTERS

First, a *quick review* of how Binary and Hexadecimal numbers are related: When the ones and zeros of four Binary bits are grouped together (from 0000 to 1111; often called a *nibble*), they can be represented by a single Hex digit (from 0 to F); both of which are used to count from 0 to 15 in Decimal. Eight Binary bits (which allow for any Decimal value from 0 to 255) are represented by **two** Hex digits; for example, the Binary number 10101010 is equal to AA in Hexadecimal. Two-digit Hex numbers (or 8-bit Binary numbers) are called **bytes**. No matter how many digits a number has in Binary or Hexadecimal, you can switch between the two by grouping and converting every four Binary bits to one Hex digit or vice versa. The Hexadecimal equivalent of a 12-bit Binary must have 3 digits (e.g., 101010111100 = ABC hex), and the largest Decimal number that can be represented by sixteen Binary bits (**65,535**) is equivalent to FFFF in Hex.

The four main (16-bit) registers of an 8086 CPU are called: The Accumulator (AX), Base (BX), Count (CX) and Data (DX) Registers. Each of these registers has a High (H) and Low (L) 8-bit half which can store one Hexadecimal byte (1 byte = 8 binary bits). Each of these high and low halves can be accessed separately by program code:

AX:	AH	AL	Accumulator
BX:	BH	BL	Base Register
CX:	CH	CL	Count Register
DX:	DH	DL	Data Register
	<-- **8** bits -->	<-- **8** bits -->	
	< **1 byte** >	< **1 byte** >	

The other 16-bit (two-byte) registers of the 8086 CPU are:

SP	Stack Pointer
BP	Base Pointer
SI	Source Index
DI	Destination Index
CS	Code Segment
DS	Data Segment
SS	Stack Segment
ES	Extra Segment
IP	Instruction Pointer

The 8086 CPU uses a method I'll call the SEGMENT: OFFSET scheme whenever it needs to access Memory locations or *Jump* from one 64kb *Segment* to another. It does this by combining two 16-bit registers together to form a 20-bit Segment:Offset value (which means it can access over 1MB of Memory this way). For certain types of tasks, the registers are often *grouped* together in the same way, but the programmer must still make sure that the CPU will use them as expected. Here are a couple tasks which are set by the CPU itself:

CS	:	IP	**Next Instruction**
SS	:	SP	**Stack Pointer**

Two other sets which are often used together in *String operations* are:

DS	:	SI	**Source Index**

ES : DI Destination Index

The "FLAGS" REGISTER

If you really want to use MS-DEBUG (or any other debugger!) to examine the operation of Assembly programs, then I advise you to copy these abbreviations onto notecards and try to understand why certain bits (flags) do/don't change in the register for each instruction in the program you're examining. (Make sure you study the final table below if you'll be reading textbooks which discuss the newer 16-bit Flags Register. Newer 32-bit debuggers, such as Borland's ***Turbo Debugger***, display the same eight Flags as DEBUG, **but** use an abbreviation for the **Name of the Flag** and simply show whether it's a 0 or 1 bit.)

```
This is an 8-bit register that holds 1-bit of data for each
of the eight "Flags" which are to be interpreted as follows:

Textbook abbrev. for Flag Name =>   of df if sf zf af pf cf
If the FLAGS were all SET (1),      -- -- -- -- -- -- -- --
they would look like this...   =>   OV DN EI NG ZR AC PE CY
If the FLAGS were all CLEARed (0),
they would look like this...   =>   NV UP DI PL NZ NA PO NC
```

FLAGS			SET (a 1-bit)	CLEARed (a 0-bit)
Overflow	of	=	OV	NV [No Overflow]
Direction	df	=	DN (decrement)	UP (increment)
Interrupt	if	=	EI (enabled)	DI (disabled)
Sign	sf	=	NG (negative)	PL (positive)
Zero	zf	=	ZR [zero]	NZ [Not zero]
Auxiliary Carry	af	=	AC	NA [No AC]
Parity	pf	=	PE (even)	PO (odd)
Carry	cf	=	CY [Carry]	NC [No Carry]

```
==============
The individual abbreviations appear to have these meanings:

OV = OVerflow, NV = No oVerflow.    DN = DowN,  UP (up).
EI = Enable Interupt, DI = Disable Interupt.

NG = NeGative, PL = PLus; a strange mixing of terms due to the
     fact that 'Odd Parity' is represented by PO (rather than
     POsitive), but they still could have used 'MI' for MInus.
```

```
ZR = ZeRo,  NZ = Not Zero.
AC = Auxiliary Carry, NA = Not Auxiliary carry.
PE = Parity Even, PO = Parity Odd.  CY = CarrY, NC = No Carry.
```

To help you understand the display in MS-DEBUG compared to any Assembly texts which make references to a 16-bit FLAGS register, here's the layout of the first 12 bits (0 - 11) of the newer 16-bit Flags Register:

```
          Symbols used by MS-DEBUG
      -----------------------------------
      OV DN EI    NG ZR    AC    PE    CY  ---> When bit = 1
      NV UP DI    PL NZ    NA    PO    NC  ---> When bit = 0

     |xx|xx|xx|--|xx|xx| 0|xx| 0|xx| 1|xx|    Newer Designations:
Bits: 11 10  9  8  7  6  5  4  3  2  1  0
       |  |  |  |  |  |  |  |  |  |  |  '---  CF - Carry Flag
       |  |  |  |  |  |  |  |  |  |  '---  is always a 1
       |  |  |  |  |  |  |  |  |  '---  PF - Parity Flag
       |  |  |  |  |  |  |  |  '---  is always a 0
       |  |  |  |  |  |  |  '---  AF - Auxiliary (Carry) Flag
       |  |  |  |  |  |  '---  is always a 0
       |  |  |  |  |  '---  ZF - Zero Flag
       |  |  |  |  '---  SF - Sign Flag
       |  |  |  '---  (TF Trap Flag - not shown in MS-DEBUG)
       |  |  '---  IF - Interrupt Flag
       |  '---  DF - Direction Flag
       '---  OF - Overflow flag
```