APPENDIX A

DEBUG PROGRAMMING

OVERVIEW

DEBUG is a program included in the MS-DOS operating systems that allows the programmer to monitor a program's execution closely for debugging purposes. Specifically, it can be used to examine and alter the contents of memory, to enter and run programs, and to stop programs at certain points in order to check or even change data. This appendix provides a tutorial introduction to the DEBUG program. You will learn how to enter and exit DEBUG, how to enter, run, and debug programs, how to examine and alter the contents of registers and memory, plus some additional features of DEBUG that prove useful in program development. Numerous examples of Assembly language programming in DEBUG are given throughout and the appendix closes with a quick reference summary of the DEBUG commands.

First, a word should be said about the examples in this appendix. Within examples, what you should type in will be represented in plain text caps:

PLAIN TEXT REPRESENTS WHAT THE USER TYPES IN

and the response of the DEBUG program will be in bold caps:

BOLD CAPS REPRESENT THE COMPUTER'S RESPONSE

The examples in this appendix assume that the DEBUG program is in drive A and that your programs are on drive B. If your system is set up differently, you will need to keep this in mind when typing in drive specifications (such as "B:"). It is strongly suggested that you type in the examples in DEBUG and try them for yourself. The best way to learn is by doing!

SECTION A.1: ENTERING AND EXITING DEBUG

To enter the DEBUG program, simply type its name at the command prompt:

```
C:\>DEBUG <return>
-
```

"DEBUG" may be typed in either uppercase or lowercase. Again let us note that this example assumes that the DEBUG program is on the disk in drive A. After "DEBUG" and the carriage return (or enter key) is typed in, the DEBUG prompt "-" will appear on the following line. DEBUG is now waiting for you to type in a command.

Now that you know how to enter DEBUG, you are ready to learn the DEBUG commands. The first command to learn is the quit command, to exit DEBUG.

The quit command, Q, may be typed in either uppercase or lowercase. This is true for all DEBUG commands. After the Q and the carriage return have been entered, DEBUG will return you to the command prompt. This is shown in Example A-1.

Example A-1 : Entering and Exiting DEBUG

```
C:\>DEBUG <return>
-Q  <return>
C:\>
```

SECTION A.2: EXAMINING AND ALTERING REGISTERS

The register command allows you to examine and/or alter the contents of the internal registers of the CPU. The R command has the following syntax:

```
R  <register name >
```

The R command will display all registers unless the optional <register name> field is entered, in which case only the register named will be displayed.

Example A-2 : Using the R Command to Display All Registers

```
C:\>DEBUG <return>
-R
AX=0000 BX=0000 CX=0000 DX=0000 SP=FFEE BP=0000 SI=0000 DI=0000
DS=15EF ES=15EF SS=15EF CS=15EF IP=0100 NV UP EI PL NZ NA PO NC
15EF:0100 0AE4           OR       AH,AH
```

After the R and the carriage return are typed in, DEBUG responds with three lines of information. The first line displays the general-purpose, pointer, and index registers' contents. The second line displays the segment registers' contents, the instruction pointer's current value, and the flag register bits. The codes at the end of line two, "NV UP DI ... NC", indicate the status of eight of the bits of the flag register. The flag register and its representation in DEBUG are discussed in Section A.6. The third line shows some information useful when you are programming in DEBUG. It shows the instruction pointed at by CS:IP. The third line on your system will vary from what is shown above. For the purpose at hand, concentrate on the first two lines. The explanation of the third line will be postponed until later in this appendix.

When you enter DEBUG initially, the general-purpose registers are set to zero and the flag bits are all reset. The contents of the segment registers will vary depending on the system you are using, but all segment registers will have the same value, which is decided by the operating system. For instance, notice in Example A-2 above that all segment registers contain 0C44H. It is strongly recommended not to change the contents of the segment registers since these values have been set by the operating system. Note: In a later section of this appendix we show how to load an Assembly language program into DEBUG. In that case the segment registers are set according to the program parameters and registers BX and CX will contain the size of the program in bytes.

If the optional register name field is specified in the R command, DEBUG will display the contents of that register and give you an opportunity to change its value. This is seen next in Example A-3.

Example A-3 : Using the R Command to Display/Modify Registers

```
(a) Modifying the contents of a register
   -R CX
   CX  0000
   :FFFF
   -R CX
   CX  FFFF
   :

(b) DEBUG pads values on the left with zero
   -R AX
   AX  0000
   :1
   -R AX
   AX  0001
   :21
   -R AX
   AX  0021
   :321
   -R AX
   AX  0321
   :4321
   -R AX
   AX  4321
   :54321
       ^ Error
(c) Entering data into the upper byte
   -R DH
   BR Error
   -R DX
   DX  0000
   :4C00
   -
```

Part (a) of Example A-3 first showed the R command followed by register name CX. DEBUG then displayed the contents of CX, which were 0000, and then displayed a colon ":". At this point a new value was typed in, and DEBUG prompted for another command with the "-" prompt. The next command verified that CX was indeed altered as requested. This time a carriage return was entered at the ":" prompt so that the value of CX was not changed.

Part (b) of Example A-3 showed that if fewer than four digits are typed in, DEBUG will pad on the left with zeros. Part (c) showed that you cannot access the upper and lower bytes separately with the R command. If you type in any digit other than 0 through F (such as in "2F0G"), DEBUG will display an error message and the register value will remain unchanged.

See Section A.6 for a discussion of how to use the R command to change the contents of the flag register.

SECTION A.3: CODING AND RUNNING PROGRAMS IN DEBUG

In the next few topics we explore how to enter simple Assembly language instructions, and assemble and run them. The purpose of this section is to familiarize the reader with using DEBUG, not to explain the Assembly language instructions found in the examples.

A, the assemble command

The assemble command is used to enter Assembly language instructions into memory.

```
A  <starting address>
```

The starting address may be given as an offset number, in which case it is assumed to be an offset into the code segment, or the segment register can be specified explicitly. In other words, "A 100" and "A CS:100" will achieve the same results. When this command is entered at the command prompt "-", DEBUG will begin prompting you to enter Assembly language instructions. After an instruction is typed in and followed by <return>, DEBUG will prompt for the next instruction. This process is repeated until you type a <return> at the address prompt, at which time DEBUG will return you to the command prompt level. This is shown in part (a) of Example A-4.

Before you type in the commands of Example A-4, be aware that one important difference between DEBUG programming and Assembly language programming is that DEBUG assumes that all numbers are in hex, whereas most assemblers assume that numbers are in decimal unless they are followed by "H". Therefore, the Assembly language instruction examples in this section do not have "H" after the numbers as they would if an assembler were to be used. For example, you might enter an instruction such as "MOV AL,3F". In an Assembly language program written for MASM, for example, this would have been typed as "MOV AL,3FH".

As you type the instructions, DEBUG converts them to machine code. If you type an instruction incorrectly such that DEBUG cannot assemble it, DEBUG will give you an error message and prompt you to try again. Again, keep in mind that the value for the code segment may be different on your machine when you run Example A-4. Notice that each time DEBUG prompts for the next instruction, the offset has been updated to the next available location. For example, after you typed the first instruction at offset 0100, DEBUG converted this to machine language, stored it in bytes 0100 to 0102, and prompted you for the next instruction, which will be stored at offset 0103. Note: Do not assemble beginning at an offset lower than 100. The first 100H (256) bytes are reserved by the operating system and should not be used by your programs. This is the reason that examples in this book use "A 100" to start assembling instructions after the first 100H bytes.

U, the unassemble command: looking at machine code

The unassemble command displays the machine code in memory along with its equivalent Assembly language instructions. The command can be given in either format

Example A-4 : Assemble, Unassemble, and Go Commands

```
(a) Assemble command
   -A 100
   103D:0100 MOV AX,1
   103D:0103 MOV BX,2
   103D:0106 MOV CX,3
   103D:0109 ADD AX,BX
   103D:010B ADD AX,CX
   103D:010D INT 3
   103D:010E
   -
(b) Unassemble command
   -U 100 10D
   103D:0100 B80100     MOV     AX,0001
   103D:0103 BB0200     MOV     BX,0002
   103D:0106 B90300     MOV     CX,0003
   103D:0109 01D8       ADD     AX,BX
   103D:010B 01C8       ADD     AX,CX
   103D:010D CC         INT     3
   -
(c) Go command
   -R
AX=0000 BX=0000 CX=0000 DX=0000 SP=CFDE BP=0000 SI=0000 DI=0000
DS=103D ES=103D SS=103D CS=103D IP=0100 NV UP DI PL NZ NA PO NC
103D:0100 B80100         MOV   AX,0001
   -G
AX=0006 BX=0002 CX=0003 DX=0000 SP=CFDE BP=0000 SI=0000 DI=0000
DS=103D ES=103D SS=103D CS=103D IP=010D NV UP DI PL NZ NA PE NC
103D:010D CC             INT   3
   -
```

shown below.

```
U <starting address >  <ending address>
U <starting address >  < L number of bytes>
```

Whereas the assemble instruction takes Assembly language instructions from the keyboard and converts them to machine code, which it stores in memory, the unassemble instruction does the opposite. Unassemble takes machine code stored in memory and converts it back to Assembly language instructions to be displayed on the monitor. Look at part (b) of Example A-4. The unassemble command was used to unassemble the code that was entered in part (a) with the assemble command. Notice that both the machine code and Assembly instructions are displayed. The command can be entered either with starting and ending addresses, as was shown in Example A-4: "U 100 10D", or with a starting address and a number of bytes in hex. The same command in the second format would be "U 100 LD", which tells DEBUG to start unassembling at CS:100 for D bytes. If the U command is entered with no addresses after it: "U <return>", then DEBUG will display 32 bytes beginning at CS:IP. Successively entering "U <return>" commands will cause DEBUG to display consecutive bytes of the program, 32 bytes at a time. This is an easy way to look through a large program.

G, the go command

The go command instructs DEBUG to execute the instructions found between the two given addresses. Its format is

```
G < = starting address>  <stop address(es)>
```

If no addresses are given, DEBUG begins executing instructions at CS:IP until a breakpoint is reached. This was done in part (c) of Example A-4. Before the instructions were executed, the R command was used to check the values of the registers. Since CS:IP pointed to the first instruction, the G command was entered, which caused execution of instructions up until "INT 3", which terminated execution. After a breakpoint is reached, DEBUG displays the register contents and returns you to the command prompt "-". Up to 10 stop addresses can be entered. DEBUG will stop execution at the first of these breakpoints that it reaches. This can be useful for programs that could take several different paths.

At this point the third line of the register dump has become useful. The purpose of the third line is to show the location, machine code, and Assembly code of the next instruction to be executed. In Example A-5, look at the last line in the register dump given after the G command. Notice, at the leftmost part of line three, the value CS:IP. The values for CS and IP match those given in lines one and two. After CS:IP is the machine code, and after the machine code is the Assembly language instruction.

Part (a) of Example A-5 is the same as part (c) of Example A-4. The go command started at CS:IP and executed instructions until it reached instruction "INT 3". Part (b) gave a starting address but no ending address; therefore, DEBUG executed instructions from offset 100 until "INT 3" was reached. This could also have been typed in as "G =CS:100". Part (c) gave both starting and ending addresses. We can see from the register

Example A-5 : Various Forms of the Go Command

```
The program is first assembled:
   -A 100
   103D:0100 MOV AX,1
   103D:0103 MOV BX,2
   103D:0106 MOV CX,3
   103D:0109 ADD AX,BX
   103D:010B ADD AX,CX
   103D:010D INT 3
   103D:010E
(a) Go command in form  "G"
   -G
AX=0006 BX=0002 CX=0003 DX=0000 SP=CFDE BP=0000 SI=0000 DI=0000
DS=103D ES=103D SS=103D CS=103D IP=010D NV UP DI PL NZ NA PE NC
103D:010D CC              INT     3
   -
(b) Go command in form "G  = start address"
   -G =100
AX=0006 BX=0002 CX=0003 DX=0000 SP=CFDE BP=0000 SI=0000 DI=0000
DS=103D ES=103D SS=103D CS=103D IP=010D NV UP DI PL NZ NA PE NC
103D:010D CC              INT     3
   -
(c) Go command form "G  = start address  endingaddress"
   -G =100 109
AX=0001 BX=0002 CX=0003 DX=0000 SP=CFDE BP=0000 SI=0000 DI=0000
DS=103D ES=103D SS=103D CS=103D IP=0109 NV UP DI PL NZ NA PE NC
103D:0109 01D8            ADD     AX,BX
   -
(d) Go command format "G   address"
   -R IP
   IP 0109
   :0100
   -G 109
AX=0001 BX=0002 CX=0003 DX=0000 SP=CFDE BP=0000 SI=0000 DI=0000
DS=103D ES=103D SS=103D CS=103D IP=0109 NV UP DI PL NZ NA PE NC
103D:0109 01D8            ADD     AX,BX
   -
```

results that it did execute from offset 100 to 109. Part (d) gave only the ending address. When the start address is not given explicitly, DEBUG uses the value in register IP. Be sure to check that value with the register command before issuing the go command without a start address.

T, the trace command: a powerful debugging tool

The trace command allows you to trace through the execution of your programs one or more instructions at a time to verify the effect of the programs on registers and/or data.

```
T <= starting address>  <number of instructions>
```

This tells DEBUG to begin executing instructions at the starting address. DEBUG will execute however many instructions have been requested in the second field. The default value is 1 if no second field is given. The trace command functions similarly to the go command in that if no starting address is specified, it starts at CS:IP. The difference between this command and the go command is that trace will display the register contents after each instruction, whereas the go command does not display them until after termination of the program. Another difference is that the last field of the go command is the stop address, whereas the last field of the trace command is the number of instructions to execute.

Example A-6 shows a trace of the instructions entered in part (a) of Example A-4. Notice the way that register IP is updated after each instruction to point to the next instruction. The third line of the register display shows the instruction pointed at by IP, that is, the next instruction to be executed. Tracing through a program allows you to examine what is happening in each instruction of the program. Notice the value of AX after each instruction in Example A-6.

The same trace as shown in Example A-6 could have been achieved with the command "-T 5", assuming that IP = 0100. Experiment with the various forms of the trace command. "T" with no starting or count fields will execute one instruction starting at CS:IP. If no first field is given, CS:IP is assumed. If no second field is given, 1 is assumed.

If you trace a large number of instructions, they may scroll upward off the screen faster than you can read them. <Ctrl-num lock > can be used to stop the scrolling temporarily. To resume the scrolling, enter any key. This works not only on the trace command, but for any command that displays information to the screen.

Example A-6 : Trace Command

```
-T=100 5
AX=0001 BX=0000 CX=0000 DX=0000 SP=CFDE BP=0000 SI=0000 DI=0000
DS=103D ES=103D SS=103D CS=103D IP=0103 NV UP DI PL NZ NA PO NC
103D:0103 BB0200         MOV      BX,0002

AX=0001 BX=0002 CX=0000 DX=0000 SP=CFDE BP=0000 SI=0000 DI=0000
DS=103D ES=103D SS=103D CS=103D IP=0106 NV UP DI PL NZ NA PO NC
103D:0106 B90200         MOV      CX,0003

AX=0001 BX=0002 CX=0003 DX=0000 SP=CFDE BP=0000 SI=0000 DI=0000
DS=103D ES=103D SS=103D CS=103D IP=0109 NV UP DI PL NZ NA PO NC
103D:0109 01D8           ADD      AX,BX

AX=0003 BX=0002 CX=0003 DX=0000 SP=CFDE BP=0000 SI=0000 DI=0000
DS=103D ES=103D SS=103D CS=103D IP=010B NV UP DI PL NZ NA PE NC
103D:010B 01C8           ADD      AX,CX

AX=0006 BX=0002 CX=0003 DX=0000 SP=CFDE BP=0000 SI=0000 DI=0000
DS=103D ES=103D SS=103D CS=103D IP=010D NV UP DI PL NZ NA PE NC
103D:010D CC             INT      3
-
```

Loading the 8-bit and 16-bit registers is shown in Example A-7.

Example A-7 : Moving Data into 8- and 16-bit Registers

```
C:>DEBUG
-R
AX=0000 BX=0000 CX=0000 DX=0000 SP=CFDE BP=0000 SI=0000 DI=0000
DS=103D ES=103D SS=103D CS=103D IP=0100 NV UP DI PL NZ NA PO NC
103D:0100 B664          MOV      DH,64
-A 100
103D:0100 MOV AL,3F
103D:0102 MOV BH,04
103D:0104 MOV CX,FFFF
103D:0107 MOV CL,BH
103D:0109 MOV CX,1
103D:010C INT 3
103D:010D
-T =100 5
AX=003F BX=0000 CX=0000 DX=0000 SP=CFDE BP=0000 SI=0000 DI=0000
DS=103D ES=103D SS=103D CS=103D IP=0102 NV UP DI PL NZ NA PO NC
103D:0102 B704          MOV      BH,04

AX=003F BX=0400 CX=0000 DX=0000 SP=CFDE BP=0000 SI=0000 DI=0000
DS=103D ES=103D SS=103D CS=103D IP=0104 NV UP DI PL NZ NA PO NC
103D:0104 B9FFFF        MOV      CX,FFFF

AX=003F BX=0400 CX=FFFF DX=0000 SP=CFDE BP=0000 SI=0000 DI=0000
DS=103D ES=103D SS=103D CS=103D IP=0107 NV UP DI PL NZ NA PO NC
103D:0107 88F9          MOV      CL,BH

AX=003F BX=0400 CX=FF04 DX=0000 SP=CFDE BP=0000 SI=0000 DI=0000
DS=103D ES=103D SS=103D CS=103D IP=0109 NV UP DI PL NZ NA PO NC
103D:0109 B90100        MOV      CX,0001

AX=003F BX=0400 CX=0001 DX=0000 SP=CFDE BP=0000 SI=0000 DI=0000
DS=103D ES=103D SS=103D CS=103D IP=010C NV UP DI PL NZ NA PO NC
103D:010C CC            INT      3
   -
```

Example A-8 is stored starting at CS:IP of 1132:0100. This logical address corresponds to physical address 11420 (11320 + 0100).

SECTION A.4: DATA MANIPULATION IN DEBUG

Next are described three DEBUG commands that are used to examine or alter the contents of memory.

```
F     the fill command fills a block of memory with data
D     the dump command displays contents of memory to the screen
E     the enter command examines/alters the contents of memory
```

F, the fill command: filling memory with data

The fill command is used to fill an area of memory with a data item. The syntax of the F command is as follows:

```
F <starting address >  <ending address>  <data>
F <starting address > <L number of bytes> <data>
```

Example A-8 : Assembling and Unassembling a Program

```
C:>DEBUG
-R
AX=0000 BX=0000 CX=0000 DX=0000 SP=CFDE BP=0000 SI=0000 DI=0000
DS=1132 ES=1132 SS=1132 CS=1132 IP=0100 NV UP DI PL NZ NA PO NC
1132:0100 BED548         MOV     SI,48D5
-A 100
1132:0100 MOV AL,57
1132:0102 MOV DH,86
1132:0104 MOV DL,72
1132:0106 MOV CX,DX
1132:0108 MOV BH,AL
1132:010A MOV BL,9F
1132:010C MOV AH,20
1132:010E ADD AX,DX
1132:0110 ADD CX,BX
1132:0112 ADD AX,1F35
1132:0115
-U 100 112
1132:0100 B057    MOV   AL,57
1132:0102 B686    MOV   DH,86
1132:0104 B272    MOV   DL,72
1132:0106 89D1    MOV   CX,DX
1132:0108 88C7    MOV   BH,AL
1132:010A B39F    MOV   BL,9F
1132:010C B420    MOV   AH,20
1132:010E 01D0    ADD   AX,DX
1132:0110 01D9    ADD   CX,BX
1132:0112 05351F  ADD   AX,1F35
-
```

This command is useful for filling a block of memory with data, for example, to initialize an area of memory with zeros. Normally, you will want to use this command to fill areas of the data segment, in which case the starting and ending addresses would be offset addresses into the data segment. To fill another segment, the register should precede the offset. For example, the first command below would fill 16 bytes, from DS:100 to DS:10F with FF. The second command would fill a 256-byte block of the code segment, from CS:100 to CS:1FF with ASCII 20 (space).

```
F 100 10F FF
F CS:100 1FF 20
```

Example A-9 demonstrates the use of the F command. The data can be a series of items, in which case DEBUG will fill the area of memory with that pattern of data, repeating the pattern over and over. For example:

```
F 100 L20 00 FF
```

The command above would cause 20 hex bytes (32 decimal) starting at DS:100 to be filled alternately with 00 and FF.

D, the dump command: examining the contents of memory

The dump command is used to examine the contents of memory. The syntax of the D command is as follows:

```
D <start address >  <end  address>
D <start address >  <L number of bytes>
```

Example A-9 : Filling and Dumping a Block of Memory

```
(a) Fill and dump commands
  C:>DEBUG
  -F 100 14F 20
  -F 150 19F 00
  -D 100 19F
103D:0100  20 20 20 20 20 20 20 20-20 20 20 20 20 20 20 20
103D:0110  20 20 20 20 20 20 20 20-20 20 20 20 20 20 20 20
103D:0120  20 20 20 20 20 20 20 20-20 20 20 20 20 20 20 20
103D:0130  20 20 20 20 20 20 20 20-20 20 20 20 20 20 20 20
103D:0140  20 20 20 20 20 20 20 20-20 20 20 20 20 20 20 20
103D:0150  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00
103D:0160  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00
103D:0170  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00
103D:0180  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00
103D:0190  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00
-
(b) Filling and dumping selected memory locations
  -F 104 10A FF
  -D 104 10A
103D:0104  FF FF FF FF-FF FF FF
  -D 100 10F
103D:0100  20 20 20 20 FF FF FF FF-FF FF FF 20 20 20 20 20
-
(c) Filling and dumping code segment memory
  -F CS:100 12F 20
  -D CS:100 12F
103D:0100  20 20 20 20 20 20 20 20-20 20 20 20 20 20 20 20
103D:0110  20 20 20 20 20 20 20 20-20 20 20 20 20 20 20 20
103D:0120  20 20 20 20 20 20 20 20-20 20 20 20 20 20 20 20
  -
```

The D command can be entered with a starting and ending address, in which case it will display all the bytes between those locations. It can also be entered with a starting address and a number of bytes (in hex), in which case it will display from the starting address for that number of bytes. If the address is an offset, DS is assumed. The D command can also be entered by itself, in which case DEBUG will display 128 consecutive bytes beginning at DS:100. The next time "D" is entered by itself, DEBUG will display 128 bytes beginning at wherever the last display command left off. In this way, one can easily look through a large area of memory, 128 bytes at a time.

Example A-10 demonstrates the use of the fill and dump commands. Part (a) shows two fill commands to fill areas of the data segment, which are then dumped. Part (b) was included to show that small areas of memory can be filled and dumped. Part (c) shows how to fill and dump to memory from other segments. Keep in mind that the values for DS and CS may be different on your machine.

E, the enter command: entering data into memory

The fill command was used to fill a block with the same data item. The enter command can be used to enter a list of data into a certain portion of memory. The syntax of the E command is as follows:

```
E <address>  <data list>
E <address>
```

Part (a) of Example A-10 showed the simplest use of the E command, entering the starting address, followed by the data. That example showed how to enter ASCII data, which can be enclosed in either single or double quotes. The E command has another powerful feature: the ability to examine and alter memory byte by byte. If the E com-

Example A-10 : Using the E Command to Enter Data into Memory

```
(a) Entering data with the E command
  -E 100 'John Snith'
  -D 100 10F
103D:0100 4A 6F 68 6E 20 53 6E 69-74 68 20 20 20 20 20 20  John Snith

(b) Altering data with the E command
  -E 106
  103D:0106  6E.6D
  -D 100 10F
103D:0100 4A 6F 68 6E 20 53 6D 69-74 68 20 20 20 20 20 20  John Smith

(c) Another way to alter data with the E command, hitting the
    space bar to go through the data a byte at a time
  -E 100
103D:0100 4A. 6F. 68. 6E. 20. 53. 6E.6D
  -D 100 10F
103D:0100 4A 6F 68 6E 20 53 6D 69-74 68 20 20 20 20 20 20  John Smith
  -

(d)  Another way to alter data with the E command
  -E 107
  103D:0107 69.-
  103D:0106 6E.6D
  -
```

mand is entered with a specific address and no data list, DEBUG assumes that you wish to examine that byte of memory and possibly alter it. After that byte is displayed, you have four options:

1. You can enter a new data item for that byte. DEBUG will replace the old contents with the new value you typed in.
2. You can press <return>, which indicates that you do not wish to change the value.
3. You can press the space bar, which will leave the displayed byte unchanged but will display the next byte and give you a chance to change that if you wish.
4. You can enter a minus sign, "-", which will leave the displayed byte unchanged but will display the previous byte and give you a chance to change it.

Look at part (b) in Example A-10. The user wants to change "Snith" to "Smith". After the user typed in "E 106", DEBUG responded with the contents of that byte, 6E, which is ASCII for n, and prompted with a ".". Then the user typed in the ASCII code for "m", 6D, entered a carriage return, and then dumped the data to see if the correction was made. Part (c) of Example A-10 showed another way to make the same correction. The user started at memory offset 100 and pressed the space bar continuously until the desired location was reached. Then he made the correction and pressed carriage return.

Finally, part (d) showed a third way the same correction could have been made. In this example, the user accidentally entered the wrong address. The address was one byte past the one that needed correction. The user entered a minus sign, which caused DEBUG to display the previous byte on the next line. Then the correction was made to that byte. Try these examples yourself since the E command will prove very useful in debugging your future programs.

The E command can be used to enter numerical data as well:

```
E 100 23 B4 02 4F
```

Example A-11 gives an example of entering code with the assemble command, entering data with the enter command, and running the program.

Example A-11 : Entering Data and Code and Running a Program

```
C:>DEBUG
-A 100
103D:0100 MOV AL,00
103D:0102 ADD AL,[0200]
103D:0106 ADD AL,[0201]
103D:010A ADD AL,[0202]
103D:010E ADD AL,[0203]
103D:0112 ADD AL,[0204]
103D:0116 INT 3
103D:0117
-E DS:0200 25 12 15 1F 2B
-D DS:0200 020F
103D:0200 25 12 15 1F 2B 02 00 E8-51 FF C3 E8 1E F6 74 03 %...+..hQ.Ch.vt.
-G =100 116
AX=0096 BX=0000 CX=0000 DX=0000 SP=CFDE BP=0000 SI=0000 DI=0000
CS=103D ES=103D SS=103D CS=103D IP=0116 OV UP DI NG NZ AC PE NC
103D:0116 CC            INT       3
-
```

SECTION A.5: EXAMINING/ALTERING THE FLAG REGISTER IN DEBUG

The discussion of how to use the R command to examine/alter the contents of the flag register was postponed until this section, so that program examples that affect the flag bits could be included. Table A-1, on the following page, gives the codes for 8 bits of the flag register, which are displayed whenever a G, T, or R DEBUG command is given.

Table A-1: Codes for the Flag Register

Flag	Code When Set (= 1)	Code When Reset (= 0)
OF overflow flag	OV (overflow)	NV (no overflow)
DF direction flag	DN (down)	UP (up)
IF interrupt flag	EI (enable interrupt)	DI (disable interrupt)
SF sign flag	NG (negative)	PL (plus, or positive)
ZF zero flag	ZR (zero)	NZ (not zero)
AF auxiliary carry flag	AC (auxiliary carry)	NA (no auxiliary carry)
PF parity flag	PE (parity even)	PO (parity odd)
CF carry flag	CY (carry)	NC (no carry)

If all the bits of the flag register were reset to zero, as is the case when DEBUG is first entered, the following would be displayed for the flag register:

NV UP DI PL NZ NA PO NC
Similarly, if all the flag bits were set to 1, the following would be seen:
OV DN EI NG ZR AC PE CY

Example A-12 shows how to use the R command to change the setting of the flag register.

Example A-12 showed how the flag register can be examined, or examined and then altered. When the R command is followed by "F", this tells DEBUG to display the contents of the flag register. After DEBUG displays the flag register codes, it prompts with

another "-" at the end of the line of register codes. At this point, flag register codes may be typed in to alter the flag register, or a simple carriage return may be typed in if no changes are needed. The register codes may be typed in any order.

Example A-12 : Changing the Flag Register Contents

```
-R F
NV UP DI PL NZ NA PO NC -DN OV NG
-R F
OV DN DI NG NZ NA PO NC -
-
```

Impact of instructions on the flag bits

Example A-13, on the following page, shows the effect of ADD instructions on the flag register. The ADD in part (a) involves byte addition. Adding 9C and 64 results in 00 with a carry out. The flag bits indicate that this was the result. Notice the zero flag is now ZR, indicating that the result is zero. In addition, the carry flag was set, indicating the carry out. The ADD in part (b) involves word addition. Notice that the sign flag was set to NG after the ADD instruction was executed. This is because the result, CAE0, in its binary form will have a 1 in bit 15, the sign bit. Since we are dealing with unsigned addition, we interpret this number to be positive CAE0H, not a negative number. This points out the fact that the microprocessor treats all data the same. It is up to the programmer to interpret the meaning of the data. Finally, look at the ADD in part (c). Adding AAAAH and 5556H gives 10000H, which results in BX = 0000 with a carry out. The zero flag indicates the zero result (BX = 0000), while the carry flag indicates that a carry out occurred.

Examples A-14 and A-15 show how to code a simple program in DEBUG, set up the desired data, and execute the program. This program includes a conditional jump that will decide whether to jump based on the value of the zero flag. This example also points out some important differences between coding a program in DEBUG and coding a program for an assembler such as MASM. First notice the JNZ instruction. If this were an Assembly language program, the instruction might be "JNZ LOOP_ADD", where the label LOOP_ADD refers to a line of code. In DEBUG we simply JNZ to the address. Another important difference is that an Assembly language program would have separate data and code segments. In Example A-14, the test data was entered at offset 0200, and consequently, BX was set to 0200 since it is being used as a pointer to the data. In an Assembly language program, the data would have been set up in the data segment and the instruction might have been "MOV BX,OFFSET DATA1" where DATA1 is the label associated with the data directive that stored the data.

Example A-13 : Observing Changes in the Flag Register

```
(a)
   C:>DEBUG
   -A 100
   103D:0100 MOV AL,9C
   103D:0102 MOV DH,64
   103D:0104 ADD AL,DH
   103D:0106 INT 3
   103D:0107
   -T 3
AX=009C BX=0000 CX=0000 DX=0000 SP=CFDE BP=0000 SI=0000 DI=0000
DS=103D ES=103D SS=103D CS=103D IP=0102 NV UP DI PL NZ NA PO NC
103D:0102 B664          MOV       DH,64

AX=009C BX=0000 CX=0000 DX=6400 SP=CFDE BP=0000 SI=0000 DI=0000
DS=103D ES=103D SS=103D CS=103D IP=0104 NV UP DI PL NZ NA PO NC
103D:0104 00F0          ADD       AL,DH

AX=0000 BX=0000 CX=0000 DX=6400 SP=CFDE BP=0000 SI=0000 DI=0000
DS=103D ES=103D SS=103D CS=103D IP=0106 NV UP DI PL ZR AC PE CY
103D:0106 CC            INT       3
   -

(b)
   -A 100
   103D:0100 MOV AX,34F5
   103D:0103 ADD AX,95EB
   103D:0106 INT 3
   103D:0107
   -T =100 2
AX=34F5 BX=0000 CX=0000 DX=6400 SP=CFDE BP=0000 SI=0000 DI=0000
DS=103D ES=103D SS=103D CS=103D IP=0103 NV UP DI PL NZ NA PO NC
103D:0103 05EB95        ADD       AX,95EB

AX=CAE0 BX=0000 CX=0000 DX=6400 SP=CFDE BP=0000 SI=0000 DI=0000
DS=103D ES=103D SS=103D CS=103D IP=0106 NV UP DI NG NZ AC PO NC
103D:0106 CC              INT       3
   -
(c)
   -A 100
   103D:0100 MOV BX,AAAA
   103D:0103 ADD BX,5556
   103D:0107 INT 3
   103D:0108
   -G =100 107
AX=34F5 BX=0000 CX=0000 DX=6400 SP=CFDE BP=0000 SI=0000 DI=0000
DS=103D ES=103D SS=103D CS=103D IP=0107 NV UP DI PL ZR AC PE CY
103D:0107 CC             INT       3
   -
```

Example A-14 : Tracing through a Program to Add 5 Bytes

```
C:>DEBUG
-A 100
103D:0100 MOV CX,05
103D:0103 MOV BX,0200
103D:0106 MOV AL,0
103D:0108 ADD AL,[BX]
103D:010A INC BX
103D:010B DEC CX
103D:010C JNZ 0108
103D:010E MOV [0205],AL
103D:0111 INT 3
103D:0112
-E 0200 25 12 15 1F 2B
-D 0200 020F
103D:0200  25 12 15 1F 2B 9A DE CE-1E F3 20 20 20 20 20 20  %...+.^n.
-G =100 111
AX=0096 BX=0205 CX=0000 DX=0000 SP=CFDE BP=0000 SI=0000 DI=0000
DS=103D ES=103D SS=103D CS=103D IP=0111 NV UP DI PL ZR NA PE NC
103D:0111 CC            INT       3
-D 0200 020F
103D:0200  25 12 15 1F 2B 96 DE CE-1E F3 20 20 20 20 20 20  %...+.^n.
```

Example A-15 : Data Transfer Program in DEBUG

```
C:>DEBUG
-A 100
103D:0100 MOV SI,0210
103D:0103 MOV DI,0228
103D:0106 MOV CX,6
103D:0109 MOV AL,[SI]
103D:010B MOV [DI],AL
103D:010D INC SI
103D:010E INC DI
103D:010F DEC CX
103D:0110 JNZ 0109
103D:0112 INT 3
103D:0113
-E 0210 25 4F 85 1F 2B C4
-D 0210 022F
103D:0210 25 4F 85 1F 2B C4 43 0C-01 01 01 00 02 FF FF FF %O..+DC.....
103D:0220 FF FF FF FF FF FF FF FF-FF FF FF FF 45 0D CA 2A ........E.J*
-G =100
AX=00C4 BX-0000 CX=0000 DX=0000 SP=CFDE BP=0000 SI=0216 DI=022E
DS=103D ES=103D SS=103D CS=103D IP=0112 NV UP DI PL ZR NA PE NC
103D:0112 CC            INT       3
-D 0210 022F
103D:0210 25 4F 85 1F 2B C4 43 0C-01 01 01 00 02 FF FF FF %O..+DC.....
103D:0220 FF FF FF FF FF FF FF FF-25 4F 85 1F 2B C4 CA 2A ....%O..+DJ*
-
```

Table A-2: Summary of DEBUG Commands

Function	Command Options
Assemble	A <starting address>
Compare	C <start address> <end address> <compare address> C <start address> <L number of bytes> <compare address>
Dump	D <start address> <end address> D <start address> <L number of bytes>
Enter	E < address> <data list> E < address>
Fill	F <start address> <end address> <data> F <start address> <L number of bytes> <data>
Go	G < = start address> <end address(es)>
Hexarith	H <number 1> <number 2>
Load	L <start address> <drive> <start sector> <sectors>
Move	M <start address> <end address> <destination> M <start address> <L number of bytes> <destination>
Name	N <filename>
Procedure	P < = start address> <number of instruction>
Register	R <register name>
Search	S <start address> <end address> <data> S <start address> <L number of bytes> <data>
Trace	T < = start address> <number of instruction>
Unassemble	U <start address> <end address> U <start address> <L number of bytes>
Write	W <start address> <drive> <start sector> <sectors>

Notes:
1. All addresses and numbers are given in hex .
2. Commands may be entered in lowercase or uppercase , or a combination.
3. Ctrl-c will stop any command.
4. Ctrl-Num Lock will stop scrolling of command output . To resume scrolling, enter any key.