APPENDIX D

DEBUG and CodeView Reference

The DEBUG and CodeView utilities are discussed in this appendix. DEBUG comes with DOS and allows the user instruction-level control over an .EXE or .COM program. CodeView does the same, with a fancier environment (multiple color windows) and with the addition of being able to handle newer 80386, 80486, and Pentium instructions. CodeView is supplied with Microsoft MASM.

Let us examine DEBUG first.

USING DEBUG TO EXECUTE AN 80x86 PROGRAM

What Is DEBUG?

DEBUG is a utility program that allows a user to load an 80x86 program into memory and execute it step by step. DEBUG displays the contents of all processor registers after each instruction executes, allowing the user to determine if the code is performing the desired task. DEBUG only displays the 16-bit portion of the general purpose registers. CodeView is capable of displaying the entire 32 bits. DEBUG is a very useful debugging tool. We will use DEBUG to step through a number of simple programs, gaining familiarity with DEBUG's commands as we do so. DEBUG contains commands that can display and modify memory, assemble instructions, disassemble code already placed into memory, trace through single or multiple instructions, load registers with data, and do much more.

DEBUG loads into memory like any other program, in the first available slot. The memory space used by DEBUG for the user program begins *after* the end of DEBUG's code. If an .EXE or .COM file were specified, DEBUG would load the program according to accepted DOS conventions.

Getting Started

The best way to get familiar with DEBUG is to work through some examples with it. The first example we will use is this three-instruction sequence:

```
MOV   AL,7
MOV   BH,2
ADD   AL,BH
```

The first instruction places the number 7 into register AL. The second instruction places 2 into register BH. These two registers are added together in the third instruction, with the results going into AL. With DEBUG, we will be able to type in the instructions as they appear. DEBUG will automatically assemble them and place their respective codes into memory. We will then be able to examine the results of each instruction by tracing through the instructions one at a time.

The first step is to invoke DEBUG. This is done with a simple command, printed here in boldface. Make sure your floppy or hard disk has a copy of DEBUG.COM installed on it, and that it is in your current directory. At the DOS command prompt, enter:

```
C> debug <cr>
```

The expression <cr> indicates that you should hit the return key. Because DEBUG is a .COM file, DOS will load it into memory and execute it. DEBUG uses a minus sign as its command prompt, so you should see a "−" appear on your display.

To get a list of all commands available with DEBUG, enter a question mark at DEBUG's command prompt and press <cr>. The command summary appears like this:

```
-?<cr>
assemble      A [address]
compare       C range address
dump          D [range]
enter         E address [list]
fill          F range list
go            G [=address] [addresses]
hex           H value1 value2
input         I port
load          L [address] [drive] [firstsector] [number]
move          M range address
name          N [pathname] [arglist]
output        O port byte
proceed       P [=address] [number]
quit          Q
register      R [register]
search        S range list
trace         T [=address] [value]
unassemble    U [range]
write         W [address] [drive] [firstsector] [number]
allocate expanded memory          XA [#pages]
deallocate expanded memory        XD [handle]
map expanded memory pages         XM [Lpage] [Ppage] [handle]
display expanded memory status    XS
-
```

We will begin with a small group of these commands to get the feel for how DEBUG is used.

To enter the three instructions we wish to execute, we need to use the *assemble* command. Entering the command letter "a" at the prompt should result in a display similar to this:

```
-a<cr>
1539:0100
```

This is the familiar CS:IP format used throughout the text. Instead of using the actual effective address, DEBUG shows us the CS value and the IP value. The 1539 address will most likely be different on your machine, because it is probably not configured like the one used to generate the DEBUG examples. The second address, 0100, should be the same. This is DEBUG telling us that it will place the first instruction into the code segment at offset 0100H. Bear in mind that DEBUG interprets *all* numbers as hexadecimal numbers.

When DEBUG begins execution, it sets the value of each processor register to a default value. The segment registers (CS, DS, ES, and SS) are all set to the beginning of free memory. This accounts for the 1539 address in the CS register at the beginning of the **a** command. The instruction pointer is always set to address 0100 and all other processor registers are set to 0000 with the exception of the stack pointer (SP), which is set to address FFEE. In addition, all processor *flags* are cleared. It is useful to know how the processor registers are initialized. This will become clear as we proceed through the example.

At this point you can enter the three instructions. If you make a typing mistake, use the backspace key to correct your errors before hitting return. You should see something similar to this on your display:

```
-a<cr>
1539:0100 mov al,7<cr>
1539:0102 mov bh,2<cr>
1539:0104 add al,bh<cr>
1539:0106 <cr>
-
```

Notice that the IP address changes after each instruction. The fourth instruction, if there were one, would begin at address 0106. Because we are entering only three instructions, hitting a return on the fourth line will terminate the assemble command and get us back to the command prompt.

To examine the code that was generated by each instruction, we use the *unassemble* command. Unassemble will show us the addresses, opcodes, and instruction mnemonics for the three instructions we have just entered. If unassemble is used without parameters it will show the next 20H bytes and their corresponding 80x86 instruction equivalents. We can shorten this display by using a range parameter, like this:

```
-u 100 104<cr>
```

This command tells DEBUG to unassemble the bytes between addresses 0100 and 0104. The resulting display looks like this:

```
1539:0100 B007    MOV  AL,07
1539:0102 B702    MOV  BH,02
1539:0104 00F8    ADD  AL,BH
```

Here, we can see that each instruction entered the required 2 bytes of machine code.

To begin execution we should examine the contents of each register. Then we will know which registers change as we step through the program. DEBUG's *register* command

TABLE D.1 Flag codes

Flag	*Set*	*Clear*
Overflow	OV	NV
Direction	DN	UP
Interrupt	EI	DI
Sign	NG	PL
Zero	ZR	NZ
Aux. Carry	AC	NA
Parity	PE	PO
Carry	CY	NC

can be used to display (and modify) any of the processor's registers. To display their contents, simply enter "r" and return. Yon should get a display similar to this:

```
AX=0000  BX=0000  CX=0000  DX=0000  SP=FFEE  BP=0000  SI=0000  DI=0000
DS=1539  ES=1539  SS=1539  CS=1539  IP=0100   NV UP EI PL NZ NA PO NC
1539:0100 B007          MOV  AL,07
```

Spend a few moments looking at the value displayed for each register. Note that all general purpose registers have been loaded with 0000. Also, the CS, DS, SS, and ES registers have all been initialized to the 1539 address we have seen earlier. IP points to address 0100, where we placed the first instruction. The end of the second line indicates the state of the flags. Table D.1 explains the meaning of each flag code. We can see that currently there is no carry, odd parity, no auxiliary carry, not zero, and plus indicated, along with enabled interrupts, up direction (for use with string operations), and no overflow. It is sometimes important to watch the changes in flags as we step through a program.

The last line of the display shows the instruction that will be executed next. Because we have not executed any instructions yet, this is our first instruction! To execute a single instruction we use DEBUG's *trace* command, abbreviated "t," and get the following display:

```
-t <cr>
AX=0007  BX=0000  CX=0000  DX=0000  SP=FFEE  BP=0000  SI=0000  DI=0000
DS=1539  ES=1539  SS=1539  CS=1539  IP=0102   NV UP EI PL NZ NA PO NC
1539:0102 B702          MOV     BH,02
```

By comparing this display with the previous one, we can determine that only the lower byte of AX has been changed (it now contains 07H). No other registers except IP have been affected. No condition codes have been changed. Isn't that what MOV AL,7 should do?

Tracing through the next two instructions should look something like this:

```
-t <cr>
AX=0007  BX=0200  CX=0000  DX=0000  SP=FFEE  BP=0000  SI=0000  DI=0000
DS=1539  ES=1539  SS=1539  CS=1539  IP=0104   NV UP EI PL NZ NA PO NC
1539:0104 00F8          ADD     AL,BH

-t <cr>
AX=0009  BX=0200  CX=0000  DX=0000  SP=FFEE  BP=0000  SI=0000  DI=0000
DS=1539  ES=1539  SS=1539  CS=1539  IP=0106   NV UP EI PL NZ NA PE NC
1539:0106 8B0EDF47      MOV     CX,[47DF]
```

As expected, the final value in AL is 09H (the sum of 7 and 2). The flag display indicates that the result of the ADD instruction changed the parity flag. All other flags kept their states.

As a point of interest, look at the instruction located at address 0106. Where did it come from? We did not enter this code during any point of our exercise. Nonetheless, DEBUG thinks this is the next instruction to be executed. The reason for this is that each of the PC's 640KB comes up in a random pattern when power is first applied. DEBUG will try to interpret these random patterns as valid 80x86 instructions, as it is doing with the MOV CX instruction.

To return to DOS, exit DEBUG by entering "q" (for *quit*) at the command prompt.

You, or your instructor, may find it necessary to save all of the work you perform while using DEBUG. This is easily accomplished by pressing the **Control-PrintScreen** button on the keyboard. This will cause all characters entered from the keyboard, and all characters output to the screen, to be *echoed* to the printer, which must be turned on and loaded with paper. This will produce a hard copy of your work and will allow you to scan the results of each instruction by watching how the registers change.

To stop echoing text to your printer, press the **Control-PrintScreen** button again. You may wish to use this feature of DOS as you work through the next few examples.

■ **EXAMPLE D.1:** What is the final value in AL after this sequence of instructions executes?

```
MOV   AL,27H
MOV   BL,37H
ADD   AL,BL
DAA
```

Use DEBUG to enter these four instructions and trace their execution. What are the machine codes for each instruction? What is the final value in AL? What changes must be made when entering the instructions with the assemble command?

Solution: The purpose of this example was to use the properties of the DAA instruction. When we add 27 to 37 we expect to get 64, the correct *decimal* answer. Using DEBUG to trace through the ADD instruction shows that AL contains 5EH. The next trace command executes the DAA instruction, which corrects the value in AL to 64H, the correct packed-decimal result. You should also see that the auxiliary carry flag has been set as a result of the DAA instruction.

The machine codes for each instruction are as follows:

```
B027    MOV  AL,27
B337    MOV  BL,37
00D8    ADD  AL,BL
27      DAA
```

The machine codes can be obtained in two ways. During a trace, the machine codes and mnemonics for each instruction are displayed after the register list. The unassemble command can also be used. Try **u 100 106** for this example and see what you get.

Also, notice that the "H" was left off the 27 and 37 operands, because DEBUG expects all numbers to be in hexadecimal form. If you include the "H" by accident, DEBUG will display an error message and wait for you to reenter the instruction. ■

■ **EXAMPLE D.2:** The following sequence of DEBUG instructions converts an integer stored in AX from a miles-per-hour (mph) value into a feet-per-second (fps) value. For example, 60 mph equals 88 fps. The conversion is accomplished by first multiplying AX by 5280 (14A0H) and then dividing the result by 3600 (0E10H).

```
MOV   BX,14A0
MUL   BX
MOV   BX,E10
DIV   BX
```

Add the necessary instructions to convert 60 mph into 88 fps. Show that the results are correct at each step in the conversion.

Solution: Because DEBUG works with hexadecimal numbers only, we must first convert 60 into hex. This gives us 3CH. Register AX must be loaded with this value prior to execution. The resulting code is as follows:

```
MOV   AX,3C
MOV   BX,14A0
MUL   BX
MOV   BX,E10
DIV   BX
```

When the instructions are entered with the assemble command and executed with trace, the contents of registers AX, BX, and DX are as follows:

```
MOV   AX,3C      ->    AX=003C  BX=0000  DX=0000
MOV   BX,14A0    ->    AX=003C  BX=14A0  DX=0000
MUL   BX         ->    AX=D580  BX=14A0  DX=0004
MOV   BX,E10     ->    AX=D580  BX=0E10  DX=0004
DIV   BX         ->    AX=0058  BX=0E10  DX=0000
```

Notice that after the MUL BX instruction, AX contains D580H and DX contains 0004H. Remember that when a 16-bit register is used in MUL, the result of the multiplication is 32 bits wide and is saved in registers DX and AX (with DX holding the upper 16 bits). Thus, the product is 4D580H, which is 316,800 decimal. Check for yourself that 316,800 equals 60 times 5,280.

After the DIV BX, the final result in AX is 0058H. When converted into decimal, we get 88. ■

■ **EXAMPLE D.3:** Give a sequence of instructions that will add up all integers from 1 to 10. The sum should be found in register BL.

Solution: One way to find the required sum is to add each integer from 1 to 10 to BL one at a time, like this:

```
SUB    BL,BL    ;set result to zero
ADD    BL,1     ;add 1 to BL
ADD    BL,2     ;add 2 to BL
.
.
.
```

```
ADD    BL,9     ;add 9 to BL
ADD    BL,0A    ;add 10 to BL
```

Perhaps a better solution can be found by using a *loop*. A loop is a technique used to repeat a group of instructions any number of times. These instructions are called *loop instructions* in general, and in this example the loop instructions need to be executed ten times. Examine the following DEBUG session:

```
-a<cr>
19DC:0100 SUB BL,BL<cr>
19DC:0102 MOV AL,0A<cr>
19DC:0104 ADD BL,AL<cr>
19DC:0106 DEC AL<cr>
19DC:0108 JNZ 104<cr>
19DC:010A <cr>
```

Register AL is used as the *loop counter*. The loop counter is decremented once each time the loop is executed. The JNZ instruction causes the processor to jump back to address 104 until AL gets to 0. A long sequence of trace commands will show how the sum builds in BL as AL keeps getting smaller. To keep from having to enter the trace commands so many times, you might try:

```
-t 1e<cr>
```

This will cause DEBUG to trace 1EH (or 30 decimal) instructions. The last two traces should be similar to these:

```
AX=0001  BX=0037  CX=0000  DX=0000  SP=FFEE  BP=0000  SI=0000  DI=0000
DS=19DC  ES=19DC  SS=19DC  CS=19DC  IP=0106   NV UP EI PL NZ NA PO NC
19DC:0106 FEC8          DEC     AL
-t

AX=0000  BX=0037  CX=0000  DX=0000  SP=FFEE  BP=0000  SI=0000  DI=0000
DS=19DC  ES=19DC  SS=19DC  CS=19DC  IP=0108   NV UP EI PL ZR NA PE NC
19DC:0108 75FA          JNZ     0104
-
```

Notice how the zero flag changes (from **NZ** to **ZR**) when AX goes from 0001 to 0000. Because the JNZ command watches the state of the zero flag, execution will now continue at address 10A. The final result in register BL is 37H, which is 55 decimal, the correct sum of all the integers between 1 and 10. ■

Calling DOS Functions from DEBUG

Now that you have a little experience using DEBUG to execute simple sequences of instructions, we can move on to more complicated applications. We will make use of three new DEBUG commands: *enter, dump,* and *proceed.* We will also use a DOS function call through INT 21H. This is a very versatile DOS function, capable of performing many different operations. For our first example, we will use the display-string function of INT 21H. This function is selected by placing 9 into register AH, the register used by INT 21H to determine which of its many functions have been selected. Display-string requires that the address of the first byte of the text string be placed into DS:DX before using INT 21H. This means that the string must be contained in the data segment pointed to by

DS and have an offset equal to the value in DX. Use the assemble command of DEBUG to enter these instructions:

```
MOV   AH,9
MOV   DX,200
INT   21
```

Notice that we do not initialize the DS register. Remember that DEBUG automatically sets DS, CS, ES, and SS to the same value. This guarantees that our text string will be placed into the current data segment area. The offset value of 200H used to initialize DX is not a special value; it just happened to be a round number. Because the machine codes for the instructions are being placed into memory around address 100H, 200H seemed a good place to put the text string.

We can enter the text string two ways. First, we will make use of a new DEBUG command called *enter.* Enter allows memory to be modified on a byte-by-byte basis, beginning at the address specified in the instruction. To load the text string "Hello!$" into memory at address 200H enter **e 200** and each individual ASCII byte for every character in the text string. You will get a display similar to this:

```
-e200 <cr>
1539:0200  66.48  6F.65  75.6C  6E.6C  64.6F  0D.21  0A.24  00.<cr>
```

The first pair of numbers are the actual address where the string is being loaded. The next number (66H) is the byte stored at location 200H. DEBUG follows it with a period and waits for you to enter the new byte value. The new value of 48H is entered, followed by the spacebar. Hitting the spacebar without entering any new value will skip over the location without changing its value. The display shows that 7 new bytes were entered. The last byte displayed (00) is not followed by anything because Return was hit to terminate the enter command. The 7 bytes entered are the ASCII values for the characters in the "Hello!$" string. The "$" character must be at the end of a text string for display-string to know where the string ends.

We can check our work with another new command: *dump.* Dump displays the bytes stored in a range of memory locations. To verify that the text string has been properly stored, do the following:

```
-d200 20f <cr>
1539:0200 48 65 6C 6C 6F 21 24 00-F6 38 53 79 6E 74 61 78 Hello!$..8Syntax
```

The dump command displays the starting address, followed by 16 bytes of data read from memory. The final part of each line of a dump display is the ASCII-equivalent characters for each of the 16 bytes. The dump display clearly shows that we entered the string correctly.

A second technique that can be used to enter strings uses the assemble command. To place a different string at address 400H, do this:

```
-a400<cr>
1539:0400 db 'Try this string too...$' <cr>
1539:0417 <cr>
```

The "db" directive stands for *define-byte,* and causes DEBUG to look up the ASCII values of any characters surrounded by single quotes. You could easily enter numeric values with db as well. If you count all of the characters in the new string, including the blanks, you should get 23. This indicates that locations 400H through 416H will be loaded with the corresponding ASCII byte values. The next possible location to put anything in is 417H, which DEBUG is already indicating.

Getting back to the example at hand, we have placed a text string into memory at address 200H and entered the instructions necessary to INT 21H's display-string function. Use the trace command until it gets to the INT 21 instruction. You should see that AH contains 09 and DX contains 0200. Unfortunately, there may be hundreds of instructions involved in the execution of INT 21H. It would be a waste of time to trace through every one of them. It would be nice if INT 21 could be treated as a single instruction by DEBUG, with all instructions of INT 21, including the final RETurn, executing by entering a single DEBUG command. Fortunately for us, DEBUG does have such a command: *proceed!* Proceed causes all INT, CALL, and REP instructions to be treated as single instructions. So, if a subroutine contains forty-five instructions, using the proceed command when the CALL instruction shows up in the trace will cause DEBUG to execute all forty-five instructions, and show the contents of each register *upon return from the subroutine!* We can use proceed to see what happens when we call INT 21. Your last DEBUG trace should look something like this:

```
AX=0900  BX=0000  CX=0000  DX=0200  SP=FFEE  BP=0000  SI=0000  DI=0000
DS=1539  ES=1539  SS=1539  CS=1539  IP=0105   NV UP EI PL NZ NA PO NC
1539:0105 CD21          INT     21
```

If the proceed command is now used, the resulting display becomes:

```
-p<cr>
Hello!
AX=0924  BX=0000  CX=0000  DX=0200  SP=FFEE  BP=0000  SI=0000  DI=0000
DS=1539  ES=1539  SS=1539  CS=1539  IP=0107   NV UP EI PL NZ NA PO NC
1539:0107 6C            DB      6C
```

You can see that the "Hello" string appeared on the screen in the current cursor location, and that DEBUG will get its next instruction from 1539:0107. Obviously, INT 21H must have done its job, or the string would not have appeared. The proceed command is very useful for tracing programs that involve DOS function calls.

■ **EXAMPLE D.4:** What must be done to display the second string, which was entered with the assemble command and the DB directive?

Solution: Because the string was placed at address 400H, the MOV DX,200 instruction must be changed to MOV DX,400. Then the entire sequence of instructions is executed again. DEBUG updates the IP register after each instruction executes, so it will be necessary to load IP with 100 again (if the first instruction is at 100). The register command can be used to do this. The following steps will set IP back to 100:

```
-r ip<cr>
IP 0107
:100<cr>
-
```

The instructions for the second string can now be traced as those for the first were, with the proceed command used when INT 21 shows up again. ■

We will finish this section with one final example using two more DOS function calls. INT 21H can also read the computer's time and date if the appropriate value is

placed into AH. To read the time, AH must contain 2CH. To read the date, AH must contain 2AH. The time and date are returned in various registers, as Example D.5 shows.

■ **EXAMPLE D.5:** INT 21H requires no register setup before it is called when we are reading only the time or date. The time is returned in the following way: CH contains hours, CL contains minutes, and DH seconds. Hundredths of seconds are returned in DL. The date is returned with AL containing the day of the week, CX the year (1980 to 2099 only), DH the month, and DL the day. Can you determine the time and date from these DEBUG trace displays?

```
Time trace:
AX=2C00  BX=0000  CX=0F1C  DX=0235  SP=FFEE  BP=0000  SI=0000  DI=0000
DS=1539  ES=1539  SS=1539  CS=1539  IP=0104   NV UP EI PL NZ NA PO NC
Date trace:
AX=2A02  BX=0000  CX=07CD  DX=0417  SP=FFEE  BP=0000  SI=0000  DI=0000
DS=1539  ES=1539  SS=1539  CS=1539  IP=0108   NV UP EI PL NZ NA PO NC
```

Solution: The values contained in CX and DX in the time trace indicate that the computer's time was 15:28:02 when INT 21H was called. The values in CX and DX in the date trace show the date to be 4/23/97. The day of the week stored in AL is 2, corresponding to Tuesday. Sunday is indicated by 0. ■

RUNNING .EXE AND .COM PROGRAMS WITH DEBUG

In this section we will examine one more command: *go*. DEBUG can automatically load an .EXE or .COM file into memory, performing all necessary relocation and initialization. The great advantage here is that we do not have to enter the program by hand. We can also use DEBUG to examine existing programs, executing portions of them to determine how they work. Microcode exploration with DEBUG can be a tremendous learning experience.

To load a program with DEBUG, include the name of the program in the command line. For example:

```
C> debug hello.exe<cr>
```

will cause DEBUG to load the HELLO.EXE code into memory. Once done, we can dump, unassemble, trace, or modify the code as we see fit. If we need only to execute the program, we issue the go command, and see the following display:

```
-g<cr>
Hello!
Program terminated normally
-
```

A nice advantage of using DEBUG to execute a newly developed program in this manner is that the program exits to DEBUG, not DOS, when completed. If we desire, we can now examine the stack area to see what values were pushed onto the stack. Or, if the program performed calculations and saved the results in the data segment, we can use dump to examine the results and verify their correctness. An example of this technique involves the

data summing program FINDAVE. Ten numbers are added and averaged. The results are then saved in the data segment (via the SUM and AVERAGE variables). The list file is included here to give an indication of what the program looks like and some idea as to where the results can be found.

```
                                ;Program FINDAVE.ASM: Find the average of ten words.
                                ;
                                        .MODEL SMALL
0000                                    .DATA
0000 0A                         COUNT   DB    10
0001 0000                       SUM     DW    ?
0003 0000                       AVERAGE DW    ?
0005 0064 00C8 012C 0190        VALUES  DW    100,200,300,400,500
     01F4
000F 03E8 07D0 0BB8                     DW    1000,2000,3000,4000,5000
     0FA0 1388

0000                                    .CODE
                                        .STARTUP
0017  8D 36 0005 R                      LEA   SI,VALUES    ;set up pointer to data
001B  B8 0000                           MOV   AX,0         ;set initial sum to zero
001E  8A 0E 0000 R                      MOV   CL,COUNT     ;set up loop counter
0022  03 04                     ADDLOOP: ADD  AX,[SI]      ;add new data item to sum
0024  83 C6 02                          ADD   SI,2         ;point to next data item
0027  FE C9                             DEC   CL           ;decrement loop counter
0029  75 F7                             JNZ   ADDLOOP      ;jump if CL is not zero
002B  A3 0001 R                         MOV   SUM,AX       ;save sum
002E  99                                CWD                ;convert sum to double-word
002F  BB 000A                           MOV   BX,10        ;prepare for division by ten
0032  F7 FB                             IDIV  BX           ;divide to find average
0034  A3 0003 R                         MOV   AVERAGE,AX   ;save average
                                        .EXIT

                                        END
```

By examining the list file, we can see that the sum will be stored at address 0001 and the average at address 0003 within the data segment. After execution by DEBUG, we can dump these locations to view the results. We must first use the unassemble command to find out where DEBUG located the data segment.

```
C> debug findave.exe<cr>
-g<cr>

Program terminated normally
-u<cr>
2B60:0000 BA632B      MOV     DX,2B63
2B60:0003 8EDA        MOV     DS,DX
2B60;0005 8CD3        MOV     BX,SS
2B60:0007 2BDA        SUB     BX,DX
2B60:0009 D1E3        SHL     BX,1
2B60:000B D1E3        SHL     BX,1
2B60:000D D1E3        SHL     BX,1
2B60:000F D1E3        SHL     BX,1
2B60:0011 FA          CLI
2B60:0012 8ED2        MOV     SS,DX
```

```
2B60:0014 03E3        ADD     SP,BX
2B60:0016 FB          STI
2B60:0017 8D361100    LEA     SI,[0011]
2B60:001B B80000      MOV     AX,0000
2B60:001E 8A0E0C00    MOV     CL,[000C]
```

The unassembly shows that the data segment begins at 2B63:0000. However, the address of VALUES loaded into SI is no longer 0005 as shown in the list file, but 0011. The linker relocated the entire data segment when it created FINDAVE.EXE from FINDAVE.OBJ. Further use of the **u** command will show that SUM and AVERAGE have the new offset addresses 000D and 000F, respectively. Now we can use the dump command to examine the results.

```
-d 2B63:0 1f
2B63:0000  0A 00 F7 FB A3 0F 00 B4-4C CD 21 00 0A 74 40 72    ........L.!..t@r
2B63:0010  06 64 00 C8 00 2C 01 90-01 F4 01 E8 03 D0 07 B8    .d...,..........
-q
```

The sum stored at address 000D is 4074H. This equates to 16,500, the correct sum for the ten data items included in the program. The average stored at address 000F is 0672H, which is the proper average of 1650. Thus, we see that DEBUG can be used to verify the operation of an .EXE file and let us know if the program is working properly.

Using Script Files with DEBUG

There are times when a new programming exercise presents a significant challenge and requires many DEBUG sessions to finally get it right. It is not difficult to imagine that typing in the same group of instructions over and over quickly leads to frustration, even when the last attempt provides the correct solution. One way to avoid having to duplicate the same work in a series of DEBUG sessions while a programming exercise is being developed is to use a *script* file. A script file contains all of the DEBUG commands and statements you might enter during a session and is created using an ordinary text editor. For example, consider the following script file:

```
a
mov al,5
mov cl,6
mul cl
sub al,12
mov bl,al

r
t 5
q
```

This script file contains ten lines. The first line contains DEBUG's **a** command. This will put DEBUG into assembly mode. The next five lines are the instructions we wish to assemble and place into memory. Line seven is a blank line and is important because it gets DEBUG out of assembly mode. Line eight contains DEBUG's **r** command. This will display the current register contents prior to execution. Line nine uses DEBUG's trace command to show the execution results of the five instructions entered earlier. Finally, line ten allows us to quit DEBUG from within the script file.

To use the script file, create it with a text editor and save it as FILENAME.SCR. Then, use the following command to allow DEBUG to access the script file:

```
DEBUG < FILENAME.SCR
```

The < symbol is a DOS function that allows the standard input device (the keyboard in this case) to be *redirected.* This means that instead of DEBUG waiting for a keystroke for each new command or statement, it will simply read it from the script file. So to DEBUG, using a script file is not any different from simply typing in all the statements really fast.

Once again, to get a printout of your DEBUG session, use the **Control-PrintScreen** function to toggle the printer prior to using the script file.

The script file previously discussed is used to solve the following equation:

$$BL = (6 \times AL) - 18$$

Note that the 18 in the equation is a 12 in instruction line five, because DEBUG uses only hexadecimal numbers.

In the script file, AL is initially loaded with the number 5. The equation predicts BL to have a value of 12. Examine the following DEBUG session (which was generated by the script file) to verify that the given instructions performed the calculation correctly.

```
-a
1CE2:0100 mov al,5
1CE2:0102 mov cl,6
1CE2:0104 mul cl
1CE2:0106 sub al,12
1CE2:0108 mov bl,al
1CE2:010A
-r
AX=0000  BX=0000  CX=0000  DX=0000  SP=FFEE  BP=0000  SI=0000  DI=0000
DS=1CE2  ES=1CE2  SS=1CE2  CS=1CE2  IP=0100   NV UP EI PL NZ NA PO NC
1CE2:0100 B005          MOV     AL,05
-t 5

AX=0005  BX=0000  CX=0000  DX=0000  SP=FFEE  BP=0000  SI=0000  DI=0000
DS=1CE2  ES=1CE2  SS=1CE2  CS=1CE2  IP=0102   NV UP EI PL NZ NA PO NC
1CE2:0102 B106          MOV     CL,06

AX=0005  BX=0000  CX=0006  DX=0000  SP=FFEE  BP=0000  SI=0000  DI=0000
DS=1CE2  ES=1CE2  SS=1CE2  CS=1CE2  IP=0104   NV UP EI PL NZ NA PO NC
1CE2:0104 F6E1          MUL     CL

AX=001E  BX=0000  CX=0006  DX=0000  SP=FFEE  BP=0000  SI=0000  DI=0000
DS=1CE2  ES=1CE2  SS=1CE2  CS=1CE2  IP=0106   NV UP EI PL NZ NA PO NC
1CE2:0106 2C12          SUB     AL,12

AX=000C  BX=0000  CX=0006  DX=0000  SP=FFEE  BP=0000  SI=0000  DI=0000
DS=1CE2  ES=1CE2  SS=1CE2  CS=1CE2  IP=0108   NV UP EI PL NZ NA PE NC
1CE2:0108 88C3          MOV     BL,AL

AX=000C  BX=000C  CX=0006  DX=0000  SP=FFEE  BP=0000  SI=0000  DI=0000
DS=1CE2  ES=1CE2  SS=1CE2  CS=1CE2  IP=010A   NV UP EI PL NZ NA PE NC
1CE2:010A 48            DEC     AX
-q
```

The final value in register BL is 0CH, which is the correct result.

A BRIEF LOOK AT CODEVIEW

CodeView is a fancier version of DEBUG that allows user control over the execution of a program being developed or examined. CodeView displays its information in different windows, such as the source window, register window, command window, etc. To switch from one window to another just press the F6 button.

The source window can display source code in the original form of the source file or in machine language and mnemonics. Pressing F3 switches the display format.

The register window displays the hexadecimal contents of all processor registers, both 16- and 32-bit. The state of each arithmetic flag is also displayed, along with the address and data of the last data access. When a program is traced instruction by instruction, the contents of any registers that change during execution are highlighted. This makes it easy to follow the progress of the program. The contents of any window can be sent to the printer or to a file (called CODEVIEW.LST by default). The register window looks like this when printed:

```
AX = 0000
BX = 0000
CX = 0000
DX = 0000
SP = 0000
BP = 0000
SI = 0000
DI = 0000
DS = 2B71
ES = 2B71
SS = 2B81
CS = 2B81
IP = 0010
FL = 0200

NV UP EI PL
NZ NA PO NC
```

A command window is provided to enable the user to enter debugging commands, in a fashion similar to that of DEBUG. Some of the more useful commands are as follows:

A	Assemble
BC	Breakpoint Clear
BD	Breakpoint Disable
BE	Breakpoint Enable
BL	Breakpoint List
BP	Breakpoint Set
E	Animate
G	Go
H	Help
I	Port Input
K	Stack Trace
L	Restart

MC	Memory Compare
MD	Memory Dump
ME	Memory Enter
MF	Memory Fill
MM	Memory Move
MS	Memory Search
N	Radix
O	Options
O	Port Output
P	Program Step
Q	Quit
R	Register
T	Trace
U	Unassemble
VM	View Memory
X	Examine Symbols

Help on every command is available online. For example, the command:

```
H VM
```

displays the help information that describes the VM command.

There are other windows that allow the user to watch the contents of a variable change as the program executes and see a display of the contents of a block of memory.

The assembler places special symbolic debugging information into an .EXE file when it is assembled with the /Zi option, as in:

```
ML /Zi FINDAVE.ASM
```

This symbolic information helps CodeView keep track of things at the source-file level. You must specify the name of an .EXE file when starting CodeView (as a command line parameter):

```
CV FINDAVE
```

CodeView does not require the .EXE extension.

CodeView is also capable of handling not just the 32-bit registers (EAX, EBX, etc.) but the newer 80386 and 80486 instructions as well. Recall that DEBUG is only capable of displaying the 16-bit register sizes, and cannot assemble or unassemble instructions other than those of the 8086.

CodeView allows data operands to be displayed in their intended format. For example, a 4-byte integer and a 4-byte real number must be interpreted and displayed differently. This is easily accomplished with the MD (or VM) command. A format specifier is used to control the data format. These specifiers are:

A	ASCII characters
B	Byte
C	Code
I	Integer (2 bytes)

IU	Integer unsigned (2 bytes)
IX	Integer hexadecimal (2 bytes)
L	Long (4-byte decimal)
LU	Long unsigned (4 byte)
LX	Long hexadecimal (4 bytes)
R	Real (4-byte floating point)
RL	Real long (8-byte floating point)
RT	Real 10-byte floating point

Altogether, CodeView offers a significant improvement over the debugging capabilities of DEBUG. It is worth the time invested in learning how to use all of CodeView's features.