A Beginner's Course In Assembly Language

by Hardin Brothers

The first computer language was assembly language, which substitutes symbols that almost look like English words for the 1's and zeros of the machine's "native" language. When you write in assembly language, you regularly bypass the application software and communicate directly with the CPU, BIOS, and MS-DOS. By using assembly language, you begin to understand what is really happening inside the computer.

Assembly language has other benefits, which are well known to professional programmers. It is fast—up to 100 times faster than Basic. It uses memory more efficiently than high-level languages, and it is usually the best language for controlling the computer's I/O devices, especially the video screen. The main drawback—an often prohibitively long programming time—is manageable if you use assembly language sparingly. In fact, the best software combines high-level code with assembly language subroutines.

Normally, to do any serious assembly language programming you have to buy a commercial assembler, which translates commands into the binary code understood by the computer. Conveniently, the Debug.COM program that comes with MS-DOS provides a window into machine-level and assembly language programming. In this tutorial, I'll show you how to go into Debug and see how the computer stores data in numerical form. Then I'll offer some short assembly language programs that illustrate concepts that apply to full-blown assemblers.

Debug Starts Here

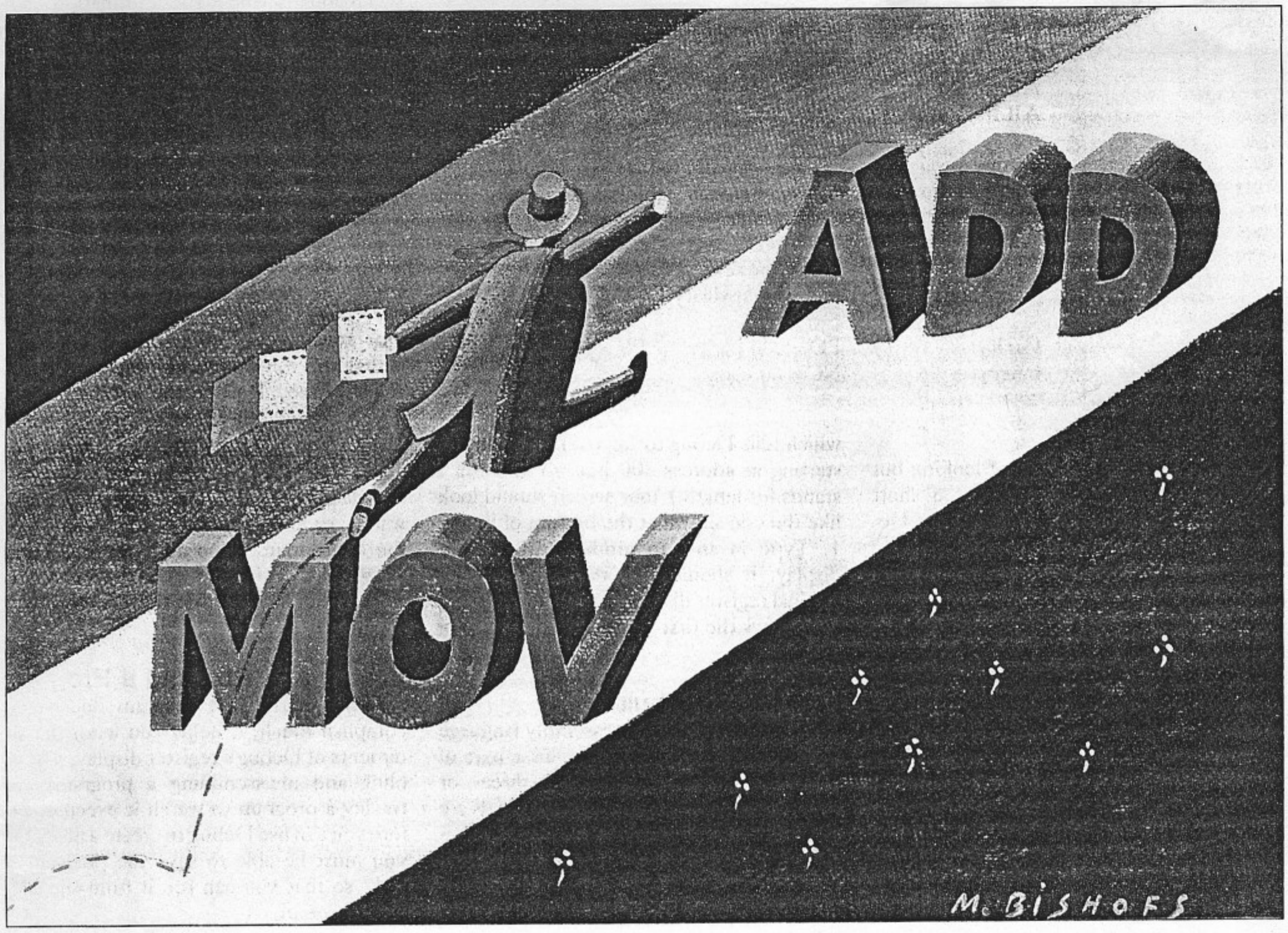
To begin, put your MS-DOS disk in drive A and type DEBUG. In a second or two, you should see a hyphen on the next line. The hyphen is Debug's prompt, and it means that Debug is waiting for your command. Debug expects you to press the enter key after each command, which you can type in either upper- or lowercase.

Type in the letter *r* and press enter. You will see a three-line display. The chart Breakdown of Initial Debug Display explains the significance of the entries. Some of the numbers and letters might be different, but the organization of the display will be essentially the same. (In all of the figures accompanying this article, the only characters you type in are those next to the Debug prompt and memory addresses [such as 16BA:0138]. Everything else is commentary or a representation of what you should see on screen.)

By typing in an *r*, you have asked Debug to show you the current contents of the CPU's registers. Registers are memory locations in the CPU (instead of in the computer's RAM). You can use some registers for almost any purpose; others are limited to specific uses. (If you don't understand how memory addresses work, see the sidebar "The Structure of Memory.")

The CPU in MS-DOS computers has 14 internal registers, each of which can hold 16 bits, or one word. The first four—AX, BX, CX, and DX—are called general-purpose registers; their use is largely up to you. Each of these four registers can also serve as two 8-bit (the equivalent of two 1-byte) registers. The top byte

WHAT YOU NEED: *MS-DOS and Basic.*



You don't need to be a pro to write programs in assembly language.

Using the MS-DOS Debug program as your starting point,

you can master the basics in record time.

of AX is called AH; the low byte is called AL. You determine when to use these areas as 16-bit registers and when to use them as 8-bit registers.

In the initial Debug display, the contents of each general-purpose register is 0000. The same value is stored in three pointer registers: BP, SI, and DI. The base pointer (BP) register generally serves as a placemark to help you manipulate a complex data structure called a stack frame.

The source index (SI) and destination index (DI) registers are mostly for moving large blocks of memory, which are called strings in assembly parlance regardless of whether they contain textual data.

The final register in the first row of the Debug display is the stack pointer (SP), an in-memory data structure that serves many purposes. Whenever a program branches to a subroutine, it stores the return address on the stack. Programmers also use the stack to temporarily store the contents of registers and sometimes to pass values from one routine to another.

The first four registers in the second row of the display are the segment registers. Because of the way memory is organized in an MS-DOS computer, two registers are required to designate memory locations. The segment registers point to a large chunk of memory; the specific address within that chunk is kept in one of the other registers.

The fifth register in the second row is the instruction pointer (IP). It always contains the address of the next instruction to be executed, in the same way that Basic always keeps track of which line number should be interpreted and executed next.

The contents of all 13 numeric registers are displayed in hexadecimal (hex) format. Assembly language programmers rarely use decimal numbers. Instead, they use either the binary (base 2) or hex (base 16) number system.

The last part of the second line in the initial Debug display contains eight two-letter abbreviations showing the state of the important bits in the flag register (see the table Status Flags). The CPU has a special 16-bit register to keep track of these status flags. They are updated to give information about the results of many assembly language operations and can be tested to determine whether a program should branch to a new set of instructions. The flags are the basis of all conditional tests in assembly language. For example, the zero flag might change from NZ to ZR to show that the result of a mathematical operation is zero. Not all flag-register bits are used.

Illustration by Maris Bishofs

Status Flags

All flags off:	NV	UP	DI	PL	NZ	NA	PO	NC
All flags on:	OV	DN	EI	NG	ZR	AC	PE	CY

Overflow:	NV = no overflow	OV = overflow
Direction:	UP = increment	DN = decrement
Interrupts:	DI = disabled	EI = enabled
Sign:	PL = plus	NG = negative
Zero:	NZ = not zero	ZR = zero
Auxiliary carry:	NA = no auxiliary carry	AC = auxiliary carry
Parity:	PO = odd parity	PE = parity even
Carry:	NC = no carry	CY = carry

Video Output

So far, you've done a lot of looking but no programming. Before writing a short program, you need to know one more Debug command. If you type in an *r*, Debug shows you the initial register display. But if you type in an *r* plus the name of a register, Debug displays the contents of the register and lets you enter a new value.

For example, if you type:

```
rax
```

Debug displays AX 0000 on one line and a colon on the next. If you enter 1111, Debug will change the contents of the AX register to 1111 hex. Remember that numbers typed into and displayed by Debug are in hex format.

Now type in the letter *a* to invoke the Assemble command. Debug answers by displaying a four-digit number, a colon, and 0100. The cursor appears without a prompt and waits for you to enter program statements.

The numbers displayed are the segment and offset of the current location in memory. The segment addresses you see will probably differ from those in the figures; they are determined by the MS-DOS version you use and whether you have any memory-resident utilities, RAM disks, or print spoolers. The number after the colon, which is called the offset, should be 0100. All .COM programs begin at address 100 hex of their memory segment, and Debug can create only .COM programs. If the offset you see is not 0100, press enter and type:

```
a100
```

to start assembly at location 0100 hex.

The first program is exceedingly simple. Type in the three instruction lines in Figure 1. Use the tab key to space between columns and press enter at the end of each line. Press the enter key once more, and you should be back at Debug's hyphen prompt. To check your work, enter:

```
u100 l 7
```

which tells Debug to "unassemble" 7 bytes starting at address 100 hex. (The letter *l* stands for length.) Your screen should look like the one shown at the bottom of Figure 1. Type in an *r* to produce the register display. It should look the same as your original register display, except that the last line shows the first instruction of your program.

A Basic Vocabulary

Every line in a Debug assembly language program has two parts. The first part of the line always contains a two-, three-, or four-letter command. These commands are called mnemonics, or memory words, because they represent exactly one CPU instruction and are easier to remember than groups of binary digits. They are sometimes called op-codes because they represent CPU operations.

After the op-code are one or two operands (or none). The number depends on the particular command and the types of information on which it operates. You may conclude the line with a semicolon followed by a remark.

The first op-code in the program is MOV, a mnemonic for the Move command. It is one of the most common mnemonics in any assembly language program. You use it to move data into registers, into memory, from register to register, and between registers and memory locations. Its name is technically incorrect, since it only *copies* information from one place to another. Like Basic's Let statement, it leaves the value in the source operand intact.

The MOV operator is always followed by two operands: the destination and the source. In the Figure 1 program, the first line tells the CPU, "Move the value 1 into the AX register." In a similar manner, the second line loads the value 2 into the BX register. The two lines are analogous to the Basic statement:

```
AX = 1: BX = 2
```

It might be obvious to you that the third line is an addition instruction; what is not so obvious is where the information originates and where the result is stored. Avoid the temptation to read the line as, "Add AX and BX." Instead, read it as, "Add the value in the BX register to that in the AX register." When you think of it that way, the expected result should be clear: The value in BX will be added to the value in AX, and the result will be left in the AX register. This line is analogous to the Basic statement:

```
AX = AX + BX
```

One advantage of programming in Debug is that you can watch the program execute step by step. Type in the letter *t* (for the Trace command) three times and watch the AX and BX registers. The Trace command tells Debug to execute an instruction and display the registers again. It lets you watch each register being loaded with the appropriate value and the final result being placed in the AX register. Your screen should resemble the one shown in Figure 2.

Saving and Running a Program

Although the first program doesn't accomplish much, it helps you learn the rudiments of Debug's register display, assembling and unassembling a program, and tracing a program to watch it execute. Before you can use Debug to create a program, you must be able to save the program to disk, so that you can run it from the MS-DOS prompt.

The program in Figure 3, while simple, introduces several new concepts. Before starting, you should clear the last program from Debug's memory. Type in the letter *q* to quit Debug, then at the MS-DOS prompt type DEBUG.

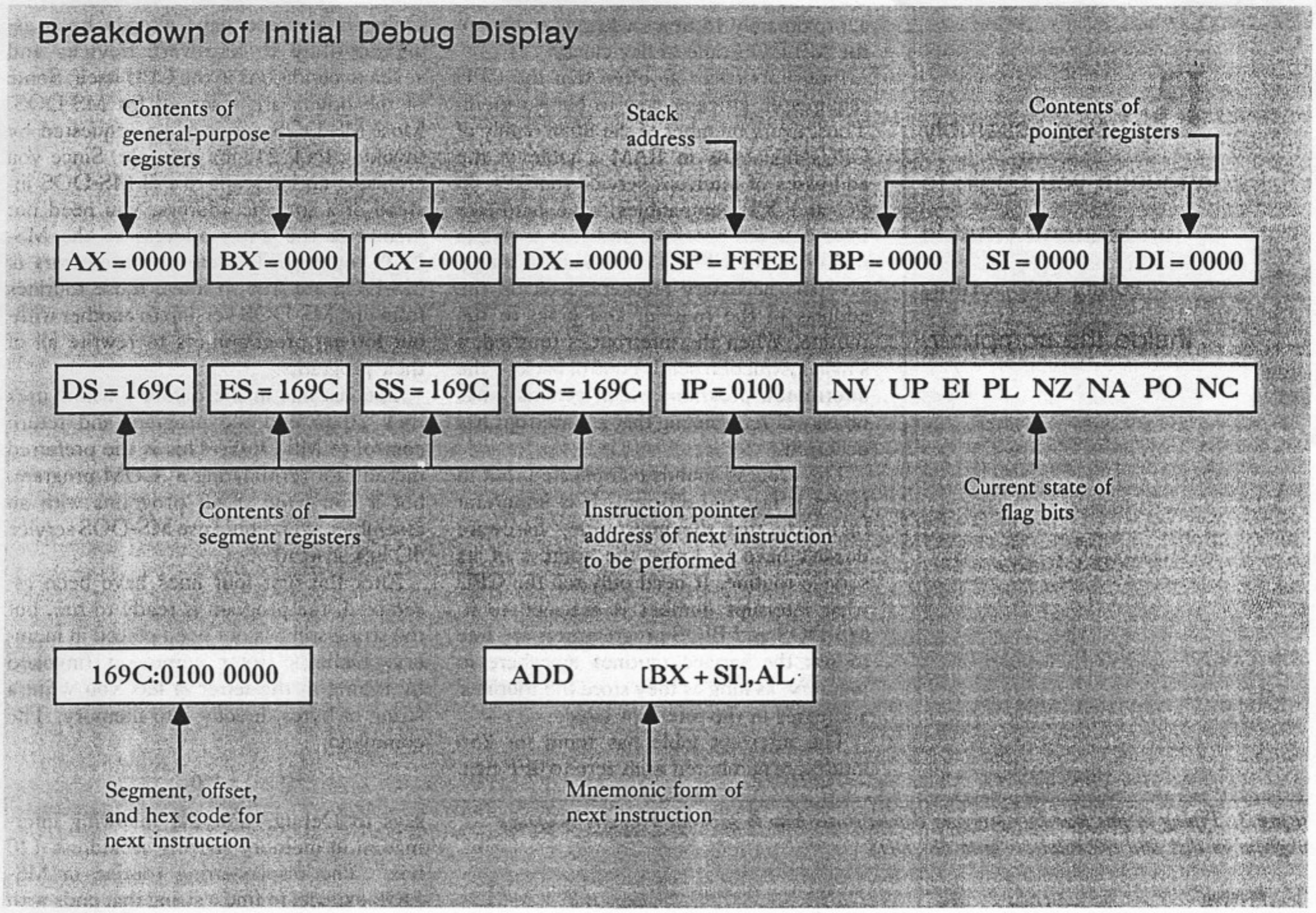
Shuffling data around in registers is a useful skill that produces no output to watch or evaluate. The Figure 3 listing, however, prints a message on screen—in this case, the phrase, "Hello world!" The program adds two new skills to your repertoire: saving a program on disk and communicating with MS-DOS.

The CPU knows nothing about video screens, disk drives, keyboards, modems, or printers. To perform I/O functions, a program must either manipulate the computer's hardware directly, which is a complicated and difficult task, or seek help from MS-DOS or the BIOS.

To write programs of your own, you need a list of MS-DOS routines and an explanation of how to use them. You can find this information in the programmer's reference manual for any MS-DOS computer. The MS-DOS routines are the same regardless of which computer you own; the BIOS routines are essentially the same for all IBM PC/XT/AT compatibles.

The first line of the program places th

Breakdown of Initial Debug Display


Contents of general-purpose registers
Stack address
Contents of pointer registers
AX = 0000
BX = 0000
CX = 0000
DX = 0000
SP = FFEE
BP = 0000
SI = 0000
DI = 0000
DS = 169C
ES = 169C
SS = 169C
CS = 169C
IP = 0100
NV UP EI PL NZ NA PO NC
Contents of segment registers
Instruction pointer address of next instruction to be performed
Current state of flag bits
169C:0100 0000
ADD [BX + SI],AL
Segment, offset, and hex code for next instruction
Mnemonic form of next instruction

value 09 hex in the AH register (the top byte of the AX register). Later, the program asks for an MS-DOS service to print a line on the screen. In general, you request each MS-DOS service by putting its number in AH, loading the other registers with the information that MS-DOS will need, and then making an MS-DOS request. In this case, you use MS-DOS service 9 (display string).

The second line loads the DX register with the value 120 hex. To display a string, MS-DOS must know where to find the string in memory, and MS-DOS service 9 expects you to put the string's address in the DS and DX registers. Since the DS register is already set to represent the memory area that the program will use, you need to set only the DX register. The value 120 hex will be past the last instruction in the program and is a convenient location to store the string.

The third line of the program invokes MS-DOS and sends it your request. The INT mnemonic stands for *interrupt*—a term that refers to the CPU's ability to be interrupted by events in the real world. Every time you press a key, the keyboard hardware interrupts the CPU, which stops what it is doing, gets the code for the key you pressed, and stores the code in its type-ahead buffer. The CPU is also interrupted

Figure 1. Type in the three program lines near the center of the figure. The simple program demonstrates how to use MOV and ADD, two common assembly language commands.

```
The program as you type it.

A>debug
-a100
169C:0100       mov     ax,1
169C:0103       mov     bx,2
169C:0106       add     ax,bx
169C:0108

The program as Debug unassembles it.

-u100 17
169C:0100 B80100        MOV     AX,0001
169C:0103 BB0200        MOV     BX,0002
169C:0106 01D8          ADD     AX,BX
-
```

End ◄

Figure 2. Debug's Trace command lets you see how each line in Figure 1 changes the CPU's registers.

```
-r
AX=0000  BX=0000  CX=0000  DX=0000  SP=FFEE  BP=0000  SI=0000  DI=0000
DS=169C  ES=169C  SS=169C  CS=169C  IP=0100   NV UP EI PL NZ NA PO NC
169C:0100 B80100        MOV     AX,0001
-t

AX=0001  BX=0000  CX=0000  DX=0000  SP=FFEE  BP=0000  SI=0000  DI=0000
DS=169C  ES=169C  SS=169C  CS=169C  IP=0103   NV UP EI PL NZ NA PO NC
169C:0103 BB0200        MOV     BX,0002
-t

AX=0001  BX=0002  CX=0000  DX=0000  SP=FFEE  BP=0000  SI=0000  DI=0000
DS=169C  ES=169C  SS=169C  CS=169C  IP=0106   NV UP EI PL NZ NA PO NC
169C:0106 01D8          ADD     AX,BX
-t

AX=0003  BX=0002  CX=0000  DX=0000  SP=FFEE  BP=0000  SI=0000  DI=0000
DS=169C  ES=169C  SS=169C  CS=169C  IP=0108   NV UP EI PL NZ NA PE NC
169C:0108 0000          ADD     [BX+SI],AL
DS:0002=00
-
```

End ◄

The best way to learn, though, is to study and experiment with short programs.

return and line feed characters with the string. The carriage return is ASCII character 13 (0D hex); the line feed is ASCII character 10 (0A hex). The Enter command tells Debug to enter the text of the message, followed by a 0D hex byte, a 0A hex byte, and the dollar sign that MS-DOS requires to terminate strings.

The first section in Figure 5 is repeated three times, each time with a different looping mechanism. The command:

```
a 130
```

tells Debug to start assembling at address 130 hex, which is the target of the JMP command at the beginning of the program.

In the first example, the DX register is loaded with the string address, the CX register is loaded with the loop count, and the AH register is loaded with the MS-DOS service number. The program calls INT 21, as before, to request that MS-DOS print the string. The next line of the program, loop 138, uses a special looping mechanism built into the CPU. The Loop instruction really tells the CPU, "Reduce the value in CX by 1. If CX is not zero, jump to the address indicated in this instruction. If CX is zero, go on to the next instruction." In other words, Debug's Loop command is much like Basic's Next statement.

The second version of the program uses a different technique to control the loop. This time, the loop counter is loaded into the BX register (you could use CX, SI, DI, or BP the same way). At the end of the loop, the instruction:

```
dec bx
```

tells the CPU to decrement (reduce by 1) the value in the BX register. The loop should continue until BX is zero, at which point the program will set the zero status flag.

The next instruction:

```
jnz 138
```

tells the CPU to jump to location 138 hex only if the zero flag is not set. Because the zero flag is set only if BX has been decremented to zero, the loop runs 20 times. These two lines are analogous to the Basic statements:

```
BX = BX - 1: IF BX <> 0 THEN GOTO 138
```

One advantage of this kind of loop is that you can nest it in another loop controlled by the CX register.

The final version of the program uses other instructions to control the loop. It loads the loop counter into the CX register again, but this time it decrements CX at the end of the loop. Then the instruction:

```
jcxz 13f
```

tells the CPU, "Jump to address 13F hex if CX contains zero. Otherwise continue to the next instruction." The following line has an unconditional jump back to the top of the loop. The CPU sees the JMP instruction only if CX has a value other than zero.

The three lines controlling the end of the loop in the third example are similar to the Basic statements:

```
CX = CX - 1: IF CX = 0 THEN GOTO 13F
ELSE GOTO 138
```

The JCXZ instruction (jump if CX is zero) is often used in complex looping structures to make a program check for the end of a loop in the middle of a block of code.

After you have assembled and saved all three versions of the program, try running them from the MS-DOS prompt to make sure they all work. If they don't, trace through them (remember to use the Go command when you come to an INT instruction) to see where you made a mistake.

You might be disappointed at the speed of the three programs. Assembly language

Video by Numbers

You can manipulate the video screen by changing the bits within the byte that controls how each character is displayed. This byte is called the screen attribute byte, and it immediately precedes the byte that contains the character code in memory. Both are stored in a special section of RAM that is set aside for video.

The program in Figure 6 prompts you to enter a screen attribute byte in hexadecimal. If you have a color monitor, consult the table below to find the 3-bit equivalents for the colors you want for the foreground and background. Insert the color values in the proper places in the byte, which are shown in the chart Screen Attributes. For the blink and high-intensity features, write 1 to turn a feature on and zero to turn it off. Then convert the entire byte to hexadecimal.

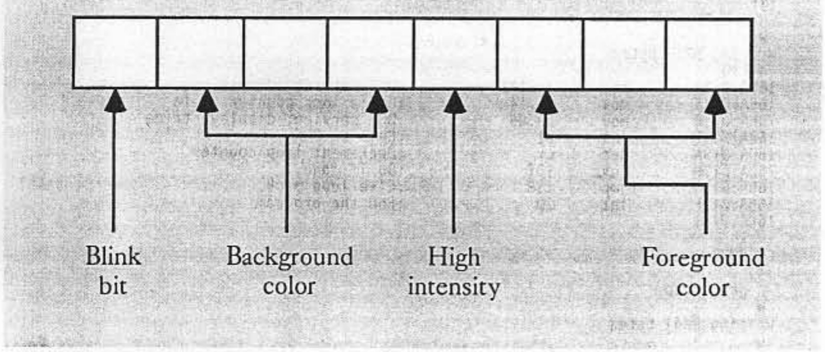
With monochrome-only graphics adapters, color value 000 is black and 111 is white. Most of the other values represent shades of gray. If the background color value is 100, displayed characters will be underlined.

With color adapters, the following color values are possible:

Color	Bit values
Black	0 0 0
Blue	0 0 1
Green	0 1 0
Cyan	0 1 1
Red	1 0 0
Magenta	1 0 1
Brown	1 1 0
Light gray	1 1 1

In this scheme, intensified black appears as dark gray, intensified brown is yellow, and intensified light gray is white. □

Screen Attributes



is supposed to be fast, yet these programs seem to run no faster than Basic's Print command. The culprit is the MS-DOS display-string service, which displays one character at a time and checks the keyboard to be sure you aren't pressing Ctrl-Break or Ctrl-C to stop the program. Most commercial programs avoid the MS-DOS display services because they are so slow.

Something Useful

The program listing in Figure 6 is much longer than the other programs, but you should be able to understand it without much difficulty. It uses some new MS-DOS services and one of the BIOS services.

The program prompts you for two hex digits and interprets them as screen attributes. It then clears the screen, sets the new attribute, and returns to MS-DOS. The new screen attributes stay in effect until another program changes them. Consult the sidebar "Video by Numbers" to learn how to manipulate bits in the screen-attribute byte and translate the result into hex.

The first step in writing a program like this with Debug is organizing the program and deciding on addresses for various parts of it. Since you need to know the addresses before typing in the program, you must guess how long each section will be. Inevitably, there will be wasted space—parts of the program won't be used at all—but the entire screen attribute program will take up only 193 bytes on disk, which is less than the minimum 1,024 bytes that MS-DOS allocates to each disk file. Thus, you don't have to worry about conserving bytes.

My outline for the program appears at the top of Figure 6. The program prints a prompt, waits for the user to enter two hex digits and a carriage return, clears the screen, sets the requested attributes, and ends. I've broken the Debug assembly process into logical blocks and added a comment to each line so you can see what is happening each step of the way.

The program begins at address 100 hex by using MS-DOS service 9 to display the prompt string. It then calls MS-DOS service 1 (read keyboard and echo) at address 107 hex to accept a keystroke. The user sees the typed character on screen but cannot backspace or edit characters.

Service 1 returns the keystroke in the AL register. Next, at address 10B hex, the program uses the Call instruction to send control to a subroutine. Debug's Call is much like Basic's Gosub instruction; when you invoke it, the current address is saved on the stack and the program jumps to the new routine. When the routine ends, it uses an RET instruction, which is analogous to Basic's Return, to go back to the main program.

The subroutine, which I'll explain in a

Figure 6. This Debug script creates an assembly language program that uses MS-DOS and BIOS services to change the display attributes.

```
Program Outline:

1ØØ: Print prompt
     Get keystroke
        Call convert routine
        If carry flag is set, start again,
           else store key value in BH
     Get keystroke
        Call convert routine
        If carry flag is set, start again,
           else add key to value in BH
     Get keystroke
        Compare to carriage return
        If different, start again
     Use Video BIOS rputine to clear screen
     End program.


15Ø: Convert routine -- Keystroke in AL converted to binary:
       If keystroke is less than 'Ø' then go to "Invalid"
       If keystroke is less than or equal to '9' then go to "Set value"
       Force keystroke to uppercase
       If keystroke is less than 'A' then go to "Invalid"
       If keystroke is greather than 'F' then go to "Invalid"
       Add 9 to key value

17Ø:   Set value:
         Erase top four bits of key value
         Reset carry flag
         Return

18Ø:   Invalid:
         Set carry flag
         Return


19Ø: Prompt message


A>debug
-a 1ØØ
169C:Ø1ØØ      mov     dx,19Ø          ;DX = string address
169C:Ø1Ø3      mov     ah,9            ;DOS service: Display String
169C:Ø1Ø5      int     21              ;Call DOS

169C:Ø1Ø7      mov     ah,1            ;DOS service: get keystroke
169C:Ø1Ø9      int     21              ;Call DOS
169C:Ø1ØB      call    15Ø             ;Convert keystroke
169C:Ø1ØE      jc      1ØØ             ;If error, start over
169C:Ø11Ø      mov     cl,4            ;Amount to shift
169C:Ø112      shl     al,cl           ;Move to top of byte
169C:Ø114      mov     bh,al           ;And save value

169C:Ø116      int     21              ;Get another keystroke
169C:Ø118      call    15Ø             ;Convert it
169C:Ø11B      jc      1ØØ             ;If error, start over
169C:Ø11D      add     bh,al           ;Else add to 1st value

169C:Ø11F      int     21              ;Get another keystroke
169C:Ø121      cmp     al,Ød           ;Carriage return?
169C:Ø123      jne     1ØØ             ;No -- start over

169C:Ø125      mov     al,Ø            ;Else scroll entire window
169C:Ø127      mov     cx,Ø            ;Ø,Ø is top-left corner
169C:Ø12A      mov     dx,184f         ;18h,4fh = 24,79 -- bottom corner
169C:Ø12D      mov     ah,6            ;BIOS service: scroll window up
169C:Ø12F      int     1Ø              ;Call Video BIOS

169C:Ø131      mov     dx,Ø            ;Ø,Ø is top-left corner
169C:Ø134      mov     bh,Ø            ;Select page Ø
169C:Ø136      mov     ah,2            ;BIOS service: set cursor position
169C:Ø138      int     1Ø              ;Call Video BIOS

169C:Ø13A      int     2Ø              ;Return to DOS
169C:Ø13C

-a 15Ø
169C:Ø15Ø      cmp     al,3Ø           ;Is key < 'Ø' ?
169C:Ø152      jb      18Ø             ;Yes -- mark as invalid
169C:Ø154      cmp     al,39           ;Is key < '9' ?
169C:Ø156      jbe     17Ø             ;Yes -- set value
169C:Ø158      and     al,df           ;Else force to upper-case
169C:Ø15A      cmp     al,41           ;Is key < 'A' ?
169C:Ø15C      jb      18Ø             ;Yes -- mark as invalid
169C:Ø15E      cmp     al,46           ;Is key > 'F' ?
169C:Ø16Ø      ja      18Ø             ;Yes -- mark as invalid
169C:Ø162      add     al,9            ;Else add offset
169C:Ø164      jmp     17Ø             ;  and set value
169C:Ø166

-a 17Ø
169C:Ø17Ø      and     al,Øf           ;Throw away top 4 bits
169C:Ø172      clc                     ;Clear error flag
```

Figure 6 continued ►

► *Figure 6 continued*

```
169C:Ø173       ret                         ;   and return
169C:Ø174

-a 18Ø
169C:Ø18Ø       stc                         ;Set error indicator
169C:Ø181       ret                         ;   and return
169C:Ø182

-e 19Ø   Ød Øa "Enter two hex digits for screen attribute ==> $"

-d 19Ø l 4Ø
169C:Ø19Ø  ØD ØA 45 6E 74 65 72 2Ø-74 77 6F 2Ø 68 65 78 2Ø   ..Enter two hex
169C:Ø1AØ  64 69 67 69 74 73 2Ø 66-6F 72 2Ø 73 63 72 65 65   digits for scree
169C:Ø1BØ  6E 2Ø 61 74 74 72 69 62-75 74 65 2Ø 3D 3D 3E 2Ø   n attribute ==>
169C:Ø1CØ  24 ØØ ØØ ØØ ØØ ØØ ØØ ØØ-ØØ ØØ ØØ ØØ ØØ ØØ ØØ ØØ   $...............

-rcx
CX ØØØØ
:cl
-n screen.com
-w
Writing ØØC1 bytes
-q
```

End ◄

moment, is set up to either convert the keystroke from ASCII to hex or to set the carry flag in the status register if the keystroke isn't in the appropriate range. Because the carry flag is easy to manipulate and test, programmers often use the carry flag to pass success-or-failure messages between routines. The line after the Call uses a JC instruction (jump only if the carry flag is set).

Generally, programmers test status flags by having the program jump to a special section of code depending on the condition of one or more of the flag bits. These conditional jumps resemble Basic's If . . . Then Goto statement. There are 30 kinds of conditional jumps in assembly language, although many are synonyms for each other. If you are tracing through a program and Debug seems to have changed the condition for jumping, don't be alarmed.

If the carry flag was not set, the ASCII-to-hex conversion subroutine was successful. In this case, the AL register contains a value between 00 hex and 0F hex to indicate which key was pressed. Since this is the first of two hex digits that the program expects the user to enter, what you really need is a value between 00 hex and 0F0 hex; that is, the program has to shift the value from the lower half of the byte to the upper half.

There are two ways to perform this shift. The program can multiply the byte in AL by 10 hex or it can shift every bit in AL four positions to the left. The second method is faster and easier to program than the first.

The instruction at address 110 hex moves a count of 4 into the CL register. The instruction:

```
shl al,cl
```

in the next line tells the CPU to shift the value in AL a number of positions left equivalent to the value in CL. During each step of the shift operation, the current value in the carry flag is discarded, the bit farthest to the left of the operand (AL, in this case) is put in the carry flag, the other bits in the operand are each moved over one place to the left, and a zero bit is inserted into the position farthest to the right of the operand. The process sounds more complicated than it is. Each shift to the left is identical to multiplying the operand by 2, so the four left shifts in the program are equivalent to multiplying by 2 to the 4th power, or 16.

After the value in AL has been shifted to the top of the byte, the result is copied to the BH register. There is a reason for selecting this particular register, which I'll explain later.

Nothing you have done so far has altered the original value of 1 in the AH register, so the program can ask for another keystroke simply by invoking INT 21 again. Once again, it calls the conversion subroutine to handle the keystroke and checks the carry flag for an error.

If the conversion subroutine doesn't report an error, the new value in AL (which is between 00 hex and 0F hex) is added to the number that was stored in BH. Then a final call to MS-DOS service 1 (at address 11F hex) gets a third keystroke, which should be a carriage return. The program then uses the CMP instruction at address 121 hex to compare the keystroke in AL with the value for a carriage return. The following line uses another conditional jump, JNE (jump if not equal), to restart the program if the user did not press enter.

Once the program gets to address 125 hex, the user has typed two hex digits followed by a carriage return, and the program has converted the digits to binary form and stored them in the BH register. It is now time to clear the screen and set the new attributes.

There are no MS-DOS services to clear the video screen, set attributes, or position the cursor. If you use MS-DOS's ANSI.SYS console driver, you can print special sequences of characters that do such things, but they tend to be slow, and they won't work if you haven't installed the ANSI.SYS file. On all IBM PC/XT/AT-compatible computers, INT 10 calls a set of BIOS ROM routines that control the video screen, change modes, position the cursor, print graphics, and so on.

Service 6 of INT 10 scrolls any portion of the active screen up a specific number of lines (service 7, which scrolls down, would serve the purpose just as well). You call it by placing the service number (6) in the AH register, placing the number of lines to scroll in AL, setting values for the top-left and bottom-right corners of the scroll window in CX and DX, and putting the attribute to be used in BH. The program has already stored the attribute in BH, so you can ignore that step for now.

At address 125 hex, the program places a zero in AL to signify that the entire window should be erased. At 127 hex, moving a zero into the CX register is the same as loading a zero into CH and CL to tell the BIOS that the top edge of the window is in row zero, column zero. (Screen rows and columns are always counted from zero, not 1.) The value placed in DH and DL should be 18 hex (24 decimal) and 4F hex (79 decimal), respectively, to indicate that the bottom edge of the window is in column 79 of row 24. Instead of using two separate instructions to place those values, the program does it in one step with the command:

```
mov dx,184F
```

Finally, the program places the service number in AH and calls INT 10.

The video BIOS routine responds by erasing the entire screen, using space characters and the new attribute byte in BH. However, it does not move the cursor to the top of the screen; you have not yet completed the assembly language equivalent of the Basic clear-screen (CLS) command.

Video BIOS service 2 positions the cursor. To call it, you must make the program place the requested cursor position in DH and DL, the video page number in BH, and the service number (2) in AH. Since you want the cursor to be in the top left of the screen, the program loads the DX register with zero. Unless another program has impolitely left the video area confused, the current video page will be zero, which the Figure 6 program places in BH. Then it calls INT 10 again to put the cursor at the top of the screen. The last step is to return to MS-DOS through INT 20.

Converting Numbers

Most of the remainder of the program in Figure 6 is concerned with converting a keystroke to a hex digit. If the user presses 5, for example, the AL register will contain 35 hex, which is the ASCII value of 5. You need a subroutine to change 35 hex to 05 hex and to make sure the user typed a valid hex digit.

The subroutine that does this begins at address 150 hex with a series of tests and conditional jumps. If the value in AL is less than 30 hex, you don't have a valid hex digit. If it is between 30 and 39 hex inclusive, it is a decimal digit and can be converted directly to binary. The JB (jump if below) instruction at address 152 hex means "Jump if the left operand was less than the right operand in the last test." The JBE operand two lines later means, "Jump if it was below or equal."

If the keystroke was not between zero and 9, it may be one of the alphabetic hex digits. It may also be in either upper- or lowercase. A look at any chart that converts ASCII characters to binary reveals only one difference between upper- and lowercase letters: Bit 5 is turned on in lowercase letters and turned off in uppercase letters. The instruction:

```
and al,df
```

at address 158 hex uses the logical And operation to check that bit 5 is turned off. The same technique works in Basic, where it is often expressed in a line like:

```
CH$ = CHR$(ASC(CH$) AND &HDF)
```

Next, the Figure 6 program performs two more tests to see if the keystroke in AL is, indeed, between the letters *A* and *F*. If it is, the program adds 9 to the character before jumping to address 170 hex. Adding the 9 is another bit of trickery (pun intended). The ASCII values for the letters *A–F* are 41–45 hex. By adding 9, you convert them to 4A–4F hex. The second digit of the resulting value is now correct.

If the character is valid, the AL register now holds a value either between 30–39 hex or 4A–4F hex. Since you want a result between 00–0F hex, you need only change the first half of the byte to zero by using another And operation at address 170 hex. Then the CLC command clears (turns off) the carry flag and an RET instruction returns control to the main part of the program.

If the user does not type a valid character, the subroutine passes control to address 180 hex. There, it turns on the carry flag with the STC (set carry flag) instruction before the program returns.

All that is left is to place the necessary prompt in memory with Debug's Enter command, display a section of memory to see how long the total program is, and save the program. (The prompt begins with a carriage return and line feed so it is always displayed on a new line, even if the user makes a mistake and the program starts over.)

After you save the program, you will undoubtedly want to return to MS-DOS and run it. If it doesn't work correctly, reenter Debug, load the program, and trace through it. (Remember not to trace through the interrupt calls.) Debugging and tracing are a necessary part of writing in assembly language, because almost every program has bugs in it at first.

What Next?

If you have enjoyed this short introduction to assembly language, you will want to experiment with your own ideas, write more complex programs, and learn to use the full CPU instruction set. To do this, you will need documentation of both the MS-DOS services and the BIOS interrupts, as well as the full CPU instruction set. There are many good books on assembly language that have both.

You will probably become frustrated with Debug.COM's limitations and want a better assembler and tracing utility. The standard assembler, and the one used in most magazine articles, is Microsoft's Macro Assembler (MASM). Newer versions of MASM have a debugging program called Symdeb, which is a large step up from Debug.COM.

The best way to learn, though, is to study and experiment with short programs. You might find that a well-commented assembly language program is at least as easy to understand as a program of similar length written in a high-level language. You will gain insight into your computer that will make you a better programmer—no matter which high-level language you choose.□

HARDIN BROTHERS *is a freelance programmer and technical writer. Write to him at 280 N. Campus Ave., Upland, CA 91786. Enclose a self-addressed, stamped envelope for a reply. You can also contact Hardin through Compuserve's Easyplex at 70007,1150.*

GLOSSARY

address—The location in memory (ROM or RAM) of a byte or word. In MS-DOS computers, each address can be represented as a unique 20-bit value. There are 1,048,576 (1MB) addresses.
bit—The smallest unit of data. It is represented in the computer as a +5 or zero voltage, and it is often written as 1 or zero. The word is a contraction of *binary digit*.
BIOS—Basic input/output system. A set of subroutines, generally in ROM, provided by the computer manufacturer for communication with the computer's hardware. DOS uses the BIOS extensively, and some BIOS routines, especially those related to the video display, are useful to programmers, as well.
byte—Eight bits grouped as a logical unit. Data is usually stored in bytes, and machine-language instructions are 1 or more bytes long.
CPU—The central processing unit, or main chip, of a personal computer. Most IBM PC/XT compatibles use the Intel 8088 or 8086 CPU. The Tandy 2000 and a few other MS-DOS computers use the 80186, and AT-type machines use the 80286. The CPU is sometimes also called the main processing unit (MPU).
DOS—Disk operating system. A series of routines that allow programmers to communicate with the disks, keyboard, screen, and other parts of the computer's hardware. The DOS is also responsible for loading, running, and ending programs.
K—Kilobyte, or 1,024 bytes. K is an abbreviation of *kilo*, which means 1,000; the actual number equals 2 to the 10th power.
math coprocessor—A special CPU designed to perform floating-point calculations at top speed. IBM PC/XT compatibles often use the Intel 8087 CPU, while AT-class computers often use the 80287.
MB—Megabyte, or 1,048,576 bytes. *Mega* means 1,000,000; the number equals 2 to the 20th power.
nibble—Half a byte, or 4 bits, that are thought of as a logical unit.
offset—The least significant 16 bits in a 20-bit memory address.
paragraph—Sixteen bytes. MS-DOS memory segments are divided into paragraphs.
RAM—Random-access memory (a misnomer). RAM is memory that can be changed by the CPU. Generally, the term refers to the memory that is allocated to MS-DOS and programs. MS-DOS computers can have a maximum of 640K or 704K of RAM, depending on the system configuration.
register—A memory location in the CPU. In the 8088 family, each register can hold one word, or 16 bits, of data. The four general-purpose registers (AX, BC, CX, and DX) can also be addressed as 8-byte registers.
ROM—Read-only memory. ROM is memory that has permanent program code "burned" into it at the factory. The CPU can execute the programs in ROM, but it cannot alter the program code. ROM contains the computer's boot-up code, BIOS routines, and, on IBM computers, part of Basic.
segment—The most significant 16 bits in a 20-bit memory address.
word—Sixteen bits, or 2 bytes.□