**DEBUG**

This tutorial is made to present an overview of the DEBUG.COM program for

the IBM PC. This utility can be extremely useful, when used correctly. It is

almost a must for Assembler Language programmers, and can also provide an

insight into the operation of the machine at the bit level. It has several

nice features, including the ability to display and change any of the registers

in the IBMPC, start and stop program execution at any time, change the program,

and look at diskettes, sector by sector. DEBUG works at the machine code

level, but it does also have the ability to disassemble machine code, and (at

dos 2.0), assemble instructions directly into machine code.

The procedure for starting DEBUG and command syntax will not be covered

here, as they are well documented in the DOS manual. What we will do is show

some examples of the various commands and the response which is expected. Note

that the segment registers will probably not be exactly what is shown. This is

normal, and should be expected.

For the examples, I will be using the demo program CLOCK.COM in the XA4

database. For those of you with the IBM assembler (MASM), the source can be

down loaded. If you do not have the assembler, or have another assembler, the

file CLOCK.HEX has been up loaded. It can be converted to a .COM file using

any of the existing HEX conversion programs on the SIG. See the file CLOCK.DOC

for more information.

STARTING DEBUG

There are two ways to start DEBUG with a file. Both ways produce the same

results, and either can be used.

In the Command Line: A>debug clock.com

Separate from the command line: A>debug

-n clock.com

-l

With either method, you will get the DEBUG prompt of a hyphen (-). DEBUG

has loaded your program and is ready to run. The description of each instruc-

tion will assume this as a starting point, unless otherwise mentioned. If at

any time you get different results, check your procedure carefully. If it is

correct, please leave me a message. I have tried to check everything, but I

have been known to make a mistake or two (anyway).

If you do have problems, you can enter the command Q (Quit) any time you

have the DEBUG prompt (-). This should return you to the DOS prompt.

RUNNING DEBUG

DISPLAY COMMANDS

**Register command**

The first thing we should look at are the registers, using the R command.

If you type in an R with no parameters, the registers should be displayed as

so:

AX=0000 BX=0000 CX=0446 DX=0000 SP=FFFE BP=0000 SI=0000 DI=0000

DS=6897 ES=6897 SS=6897 CS=6897 IP=0100 NV UP DI PL NZ NA PE NC

6897:0100 E96B01 JMP 026E

CX contains the length of the file (0446h or 1094d). If the file were

larger than 64K, BX would contain the high order of the size. This is very

important to remember when using the Write command, as this is the size of the

file to be written. Remember, once the file is in memory, DEBUG has no idea

how large the file is, or if you may have added to it. The amount of data to

be written will be taken from the BX and CX registers.

If we want to change one of the registers, we enter R and the register

name. Let's place 1234 (hexadecimal) in the AX register:

-R AX R and AX register

AX 0000 Debug responds with register and contents

: 1234 : is the prompt for entering new contents. We respond 1234

- Debug is waiting for the next command.

Now if we display the registers, we see the following:

AX=1234 BX=0000 CX=0446 DX=0000 SP=FFFE BP=0000 SI=0000 DI=0000

DS=6897 ES=6897 SS=6897 CS=6897 IP=0100 NV UP DI PL NZ NA PE NC

6897:0100 E96B01 JMP 026E

Note that nothing has changed, with the exception of the AX register. The new

value has been placed in it, as we requested. One note. The Register command

can only be used for 16 bit registers (AX, BX, etc.). It cannot change the 8

bit registers (AH, AL, BH, etc.). To change just AH, for instance, you must

enter the the data in the AX register, with your new AH and the old AL values.

**Dump command**

One of the other main features of DEBUG is the ability to display areas of

storage. Unless you are real good at reading 8088 machine language, the Dump

command is mostly used to display data (text, flags, etc.). To display code,

the Unassemble command below is a better choice. If we enter the Dump command

at this time, DEBUG will default to the start of the program. It uses the DS

register as it's default, and, since this is a .COM file, begins at DS:0100.

It will by default display 80h (128d) bytes of data, or the length you specify.

The next execution of the Dump command will display the following 80h bytes,

and so on. For example, the first execution of D will display DS:0100 for 80h

bytes, the next one DS:0180 for 80h bytes, etc. Of course, absolute segment

and segment register overrides can be used, but only hex numbers can be used

for the offset. That is, D DS:BX is invalid.

With our program loaded, if we enter the Dump command, we will see this:

6897:0100 E9 6B 01 43 4C 4F 43 4B-2E 41 53 4D 43 6F 70 79 ik.CLOCK.ASMCopy

6897:0110 72 69 67 68 74 20 28 43-29 20 31 39 38 33 4A 65 right (C) 1983Je

6897:0120 72 72 79 20 44 2E 20 53-74 75 63 6B 6C 65 50 75 rry D. StucklePu

6897:0130 62 6C 69 63 20 64 6F 6D-61 69 6E 20 73 6F 66 74 blic domain soft

6897:0140 77 61 72 65 00 00 00 00-00 00 00 00 00 00 00 00 ware............

6897:0150 00 00 00 00 00 00 00 00-00 24 00 00 00 00 00 00 .........$......

6897:0160 00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00 ................

6897:0170 00 00 00 00 00 00 00 00-00 00 00 00 44 4F 53 20 ............DOS

Notice that the output from the Dump command is divided into three parts.

On the left, we have the address of the first byte on the line. This is in the

format Segment:Offset.

Next comes the hex data at that location. Debug will always start the

second line at a 16 byte boundary; that is, if you entered D 109, you would get

7 bytes of information on the first line (109-10F), and the second line would

start at 110. The last line of data would have the remaining 9 bytes of data,

so 80h bytes are still displayed.

The third area is the ASCII representation of the data. Only the standard

ASCII character set is displayed. Special characters for the IBMPC are not

displayed; rather periods (.) are shown in their place. This makes searching

for plain text much easier to do.

Dump can be used to display up to 64K bytes of data, with one restriction:

It cannot cross a segment boundary. That is, D 0100 l f000 is valid (display

DS:0100 to DS:F0FF), but D 9000 l 8000 is not (8000h +9000h = 11000h and

crosses a segment boundary).

Since 64K is 10000h and cannot fit into four hex characters, Dump uses

0000 to indicate 64K. To display a complete segment, enter D 0 l 0. This will

display the total 64K segment.

If, at any time you want to suspend the display of data, Cntl-NumLock

works as usual. If you want to terminate the display, Cntl-Break will stop it

and return you to the DEBUG prompt.

**Search**

Search is used to find the occurrence of a specific byte or series of

bytes within a segment. The address parameters are the same as for the Dump

command, so we will not duplicate them here. However, we also need the data to

be searched for. This data can be entered as either hexadecimal or character

data. Hexadecimal data is entered as bytes, with a space or a comma as the

separator. Character data is enclosed by single or double quotes. Hex and

character data can be mixed in the same request, i.e. S 0 l 100 12 34 'abc' 56

is valid, and requests a search from DS:0000 through DS:00FF for the sequence

of 12h 34h a b c 56h, in that order. Upper case characters are different than

lower case characters, and a match will not be found if the case does not

match. For instance, 'ABC' is not the same as 'abc' or 'Abc' or any other

combination of upper and lower case characters. However, 'ABC' is identical to

"ABC", since the single and double quotes are separators only.

An example is looking for the string 'Sat'. Here's what would happen:

-S 0 l 0 'Sat'

6897:0235

-

Again, the actual segment would be different in your system, but the offset

should be the same. If we then displayed the data, we would find the string

'Saturday' at this location. We could also search on 'turda', or any other

combination of characters in the string. If we wanted to find every place we

did an Int 21h (machine code for Int is CD), we would do the following:

-S 0 l 0 cd 21

6897:0050

6897:0274

6897:027F

6897:028B

6897:02AD

6897:02B4

6897:0332

6897:0345

6897:034C

6897:043A

6897:0467

6897:047A

6897:0513

6897:0526

6897:0537

6897:0544

-

DEBUG found the hex data CD 21 at the above locations. This does not mean that

all these addresses are INT 21's, only that that data was there. It could (and

most likely is) an instruction, but it could also be an address, the last part

of a JMP instruction, etc. You will have to manually inspect the code at that

area to make sure it is an INT 21. (You don't expect the machine to do every-

thing, do you?).

**Compare command**

Along the same lines of Dump and Search commands, we have the Compare

command. Compare will take two blocks of memory and compare them, byte for

byte. If the two addresses do not contain the same information, both addresses

are displayed, with their respective data bytes. As an example, we will com-

pare DS:0100 with DS:0200 for a length of 8.

-d 0100 l 8 0200

6897:0100 E9 65 6897:0200

6897:0101 6B 70 6897:0201

6897:0102 01 74 6897:0202

6897:0103 43 65 6897:0203

6897:0104 4C 6D 6897:0204

6897:0105 4F 62 6897:0205

6897:0106 43 65 6897:0206

6897:0107 4B 72 6897:0207

None of the eight bytes compared, so we got output for each byte. If we

had gotten a match on any of the bytes, DEBUG would have skipped that byte. if

all of the locations requested matched, DEBUG would have simply responded with

another prompt. No other message is displayed. This is useful for comparing

two blocks of data from a file, or a program with the BIOS ROM. Otherwise, I

have not found a great amount of use for it.

**Unassemble command**

For debugging, one of the main commands you will use is the Unassemble

command. This command will take machine code and convert it to instructions.

Addressing is the same as for previous commands with one exception: Since we

are now working with code (the previous commands are mainly for data), the

default register is the CS register. In a .COM program, this makes very little

difference, unless you reset the DS register yourself. However, in a .EXE

file, it can make a lot of difference, as the CS and DS registers are set to

different values.

Unassemble data can lead to some interesting results. For instance, in

our example, CS:IP is set to 6897:0100. If we look at the program, we see a

JMP as the first instruction, followed by data. If we just enter U, we will

start at CS:IP (6897:0100) and start unassembling data. What we will get is a

good instruction, followed by more or less nonsense. For instance:

-U

6897:0100 E96B01 JMP 026E

6897:0103 43 INC BX

6897:0104 4C DEC SP

6897:0105 4F DEC DI

6897:0106 43 INC BX

6897:0107 4B DEC BX

And so on, through 6897:011D. We know the INC BX, DEC SP, etc. are not valid

instructions, but DEBUG doesn't, so we do have to look at the code. After

working with DEBUG a little, you will be able to spot code versus data with the

Unassemble command. For now, suffice to say that the first instruction will

take us to CS:026E and we can start from there.

If we Unassemble CS:026E, we will find something which looks a little more

like what we expect. We get:

-U 26E

6897:026E 8D167802 LEA DX,[0278]

6897:0272 B409 MOV AH,09

6897:0274 CD21 INT 21

6897:0276 EB05 JMP 027D

6897:0278 1B5B32 SBB BX,[BP+DI+32]

6897:027B 4A DEC DX

6897:027C 24B4 AND AL,B4

6897:027E 30CD XOR CH,CL

6897:0280 213C AND [SI],DI

6897:0282 027D0A ADD BH,[DI+0A]

6897:0285 8D167C01 LEA DX,[017C]

6897:0289 B409 MOV AH,09

6897:028B CD21 INT 21

6897:028D CD20 INT 20

The first few instructions look fine. But, after the JMP 027D, things

start to look a little funny. Also, note that there is no instruction starting

at 027D. We have instructions at 027C and 027E, but not 027D. This is again

because DEBUG doesn't know data from instructions. At 027C, we should (and do)

have the end of our data. But, this also translates into a valid AND instruc-

tion, so DEBUG will treat it as such. If we wanted the actual instruction at

027D, we could enter U 027D and get it, but from here, we don't know what it

is. what I'm trying to say is, DEBUG will do what ever you tell it. If you

tell it to Unassemble data, it will do so to the best of its ability. So, you

have to make sure you have instructions where you think you do.

DATA ENTRY COMMANDS

**Enter**

The Enter command is used to place bytes of data in memory. It has two

modes: Display/Modify and Replace. The difference is in where the data is

specified - in the Enter command itself, or after the prompt.

If you enter E address alone, you are in display/modify mode. DEBUG will

prompt you one byte at a time, displaying the current byte followed by a

period. At this time, you have the option of entering one or two hexadecimal

characters. If you hit the space bar, DEBUG will not modify the current byte,

but go on to the next byte of data. If you go too far, the hyphen (-) will

back up one byte each time it is pressed.

E 103

6897:0103 43.41 4C.42 4F.43 43. 4B.45

6897:0108 2E.46 41.40 53.-

6897:0109 40.47 53.

In this example, we entered E 103. DEBUG responded with the address and the

information at that byte (43). We entered the 41 and DEBUG automatically

showed the next byte of data (4C). Again, we entered 42, debug came back. The

next byte was 4F, we changed it to 43. At 106, 43 was fine with us, so we just

hit the space bar. DEBUG did not change the data, and went on to the following

bytes. After entering 40 at location 109, we found we had entered a bad value.

The hyphen key was pressed, and DEBUG backed up one byte, displaying the

address and current contents. Note that it has changed from the original value

(41) to the value we typed in (40). We then type in the correct value and

terminate by pressing the ENTER key.

As you can see, this can be very awkward, especially where large amounts

of data are concerned. Also, if you need ASCII data, you have to look up each

character and enter its hex value. Not easy, to be sure. That's where the

Replace mod of operation comes in handy. Where the Display/Modify mode is

handy for changing a few bytes at various offsets, the Replace mode is for

changing several bytes of information at one time. Data can be entered in

hexadecimal or character format, and multiple bytes can be entered at one time

without waiting for the prompt. If you wanted to store the characters 'My

name' followed by a hexadecimal 00 starting at location 103, you would enter:

E 103 'My name' 0

As in the Search command, data can be entered in character (in quotes) or

hexadecimal forms and can be mixed in the same command. This is the most

useful way of entering large amounts of data into memory.

**Fill**

The Fill command is useful for storing a lot of data of the same data. It

differs from the Enter command in that the list will be repeated until the

requested amount of memory is filled. If the list is longer than the amount of

memory to be filled, the extra items are ignored. Like the Enter command, it

will take hexadecimal or character data. Unlike the Enter command, though,

large amounts of data can be stored without specifying every character. As an

example, to clear 32K (8000h) of memory to 00h, you only need to enter:

F 0 L 8000 0

Which translates into Fill, starting at DS:0000 for a Length of 32K (8000) with

00h. If the data were entered as '1234', the memory would be filled with the

repeating string '123412341234', etc. Usually, it is better to enter small

amounts of data with the Enter command, because an error in the length parame-

ter of the Fill command can destroy a lot of work. The Enter command, however,

will only change the number of bytes actually entered, minimizing the effects

of a parameter error.

**Move**

The Move command does just what it says - it moves data around inside the

machine. It takes bytes from with the starting address and moves it to the

ending address. If you need to add an instruction into a program, it can be

used to make room for the instruction. Beware, though. Any data or labels

referenced after the move will not be in the same place. Move can be used to

save a part of the program in free memory while you play with the program, and

restore it at any time. It can also be used to copy ROM BIOS into memory,

where it can be written to a file or played with to your heart's content. You

can then change things around in BIOS without having to worry about programming

a ROM.

M 100 L 200 ES:100

This will move the data from DS:0100 to DS:02FF (Length 200) to the address

pointed to by ES:0100. Later, if we want to restore the data, we can say:

M ES:100 L 200 100

which will move the data back to its starting point. Unless the data has been

changed while at the temporary location (ES:0100), we will restore the data to

its original state.

**Assemble**

I purposely left the Assemble command to the end, as it is the most complex of

the data entry commands. It will take the instructions in the assembler lan-

guage and convert them to machine code directly. Some of the things it can't

do, however, are: reference labels, set equates, use macros, or anything else

which cannot be translated to a value. Data locations have to be referenced by

the physical memory address, segment registers, if different from the defaults,

must be specified, and RET instructions must specify the type (NEAR or FAR) of

return to be used. Also, if an instruction references data but not registers

(i.e. Mov [278],5), the Byte ptr or Word ptr overrides must be specified. One

other restriction: To tell DEBUG the difference between moving 1234h into AX

and moving the data from location 1234 into AX, the latter is coded as Mov

AX,[1234], where the brackets indicate the reference is an addressed location.

The differences between MASM and DEBUG are as follows:

MASM DEBUG Comments

Mov AX,1234 Mov AX,1234 Place 1234 into AX

Mov AX,L1234 Mov AX,[1234] Contents of add. 1234 to AX

Mov AX,CS:1234 CS:Mov AX,[1234] Move from offset of CS.

Movs Byte ptr ... Movesb Move byte string

Movs Word ptr ... Movsw Move word string

Ret Ret Near return

Ret Retf Far return

Also, Jmp instructions will be assembled automatically to Short, Near, or Far

Jmps. However, the Near and Far operands can be used to override the displace-

ment if you do need them. Let's try a very simple routine to clear the screen.

-A 100

6897:0100 mov ax,600

6897:0103 mov cx,0

6897:0106 mov dx,184f

6897:0109 mov bh,07

6897:010B int 10

6897:010D int 20

6897:010F

-

We are using BIOS interrupt 10h, which is the video interrupt. (If you

would like more information on the interrupt, there is a very good description

in the Technical Reference Manual.) We need to call BIOS with AX=600, BH=7,

CX=0, and DX=184Fh. First we had to load the registers, which we did at in the

first four instructions. The statement at offset 6897:010B actually called

BIOS. The INT 20 at offset 010D is for safety only. We really don't need it,

but with it in, the program will stop automatically. Without the INT 20, and

if we did not stop, DEBUG would try and execute whatever occurs at 010F. If

this happens to be a valid program (unlikely), we would just execute the

program. Usually, though, we will find it to be invalid, and will probably

hang the system, requiring a cntl-alt-del (maybe) or a power-off and on again

(usually). So, be careful and double check your work!

Now, we need to execute the program. To do this, enter the G command, a G

followed by the enter key. If you have entered the program correctly, the

screen will clear and you will get a message "Program terminated normally".

(More on the Go command later).

Again, I cannot stress the importance of checking your work when using the

Assemble command. The commands may assemble correctly, but cause a lot of

problems. This is especially important for the Jmp and Call commands; since

they cause an interruption in the flow of the program, they can cause the

program to jump into the middle of an instruction, causing VERY unpredictable

results.

I/O COMMANDS

**Name**

The Name command has just one purpose - specifying the name of a file which

DEBUG is going to Load or Write. It does nothing to change memory or execute a

program, but does prepare a file control block for DEBUG to work with. If you

are going to load a program, you can specify any parameters on the same line,

just like in DOS. One difference is, the extension MUST be specified. The

default is no extension. DEBUG will load or write any file, but the full file

name must be entered.

-n chkdsk.com /f

This statement prepares DEBUG for loading the program CHKDSK.COM passing the /f

switch to the program. When the Load (see below) command is executed, DEBUG

will load CHKDSK.COM and set up the parameter list (/f) in the program's input

area.

**Load**

The Load command has two formats. The first one will load a program which

has been specified by the Name command into storage, set the various registers,

and prepare for execution. Any program parameters in the Name command will be

set into the Program Segment Prefix, and the program will be ready to run. If

the file is a .HEX file, it is assumed to have valid hexadecimal characters

representing memory values, two hexadecimal characters per byte. Files are

loaded starting at CS:0100 or at the address specified in the command. For

.COM. .HEX and .EXE files, the program will be loaded, the registers set, and

CS:IP set to the first instruction in the program. For other files, the

registers are undetermined, but basically, the segment registers are set to the

segment of the PSP (100h bytes before the code is actually loaded), and BX and

CX are set to the file length. Other registers are undetermined

-n clock.com

-l

This sequence will load clock.com into memory, set IP to the entry point of

0100, and CX will contain 0446, the hexadecimal size of the file. The program

is now ready to run.

The second form of the Load command does not use the Name command. It is

used to load absolute sectors from the disk (hard or soft) into memory. The

sector count starts with the first sector of track 0 and continuing to the end

of the track. The next sector is track 0, second side (if double sided), and

continues to the end of that sector. Then, back to the first side, track 1,

and so on, until the end of the disk. Up to 80h (128d) sectors can be loaded

at one time. To use, you must specify starting address, drive (0=A, 1=B,

etc.), starting sector, and number of sectors to load.

-l 100 0 10 20

This instruction tells DEBUG to load, starting at DS:0100, from drive A, sector

10h for 20h sectors. DEBUG can sometimes be used this way to recover part of

the information on a damaged sector. If you get an error, check the memory

location for that data. Often times, part of the data has been transferred

before the error occurs and the remainder (especially for text files) can be

manually entered. Also, repetitive retrys will sometimes get the information

into memory. This can then be rewritten on the same diskette (see the Write

command below), or copied to the same sector on another diskette. In this way,

the data on a damaged disk can sometimes be recovered.

**Write**

The write command is very similar to the Load command. Both have two

modes of operation, and both will operate on files or absolute sectors. As you

have probably guessed, the Write command is the opposite of the Load command.

Since all the parameters are the same, we will not cover the syntax in detail.

However, one thing worth mentioning: When using the file mode of the Write

command, the amount of data to be written is specified in BX and CX, with BX

containing the high-order file size. The start address can be specified or is

defaulted to CS:0100. Also, files with an extension of .EXE or .HEX cannot be

written out, and error message to that effect will be displayed. If you do

need to change a .EXE or .HEX file, simply rename and load it, make your

changes, save it and name it back to its original filename.

**Input**

The Input command can be used to read a byte of data from any of the I/O

ports in the PC. The port address can be either a one or two byte address.

DEBUG will read the port, and display the contents.

-i 3fd

7D

-

This is the Line input port for the first Asynchronous adapter. Your data may

be different, as it depends on the current status of the port. It indicates

the data in the register at the time it was read was 7Dh. Depending on the

port, this data may change, as the ports are not controlled by the PC.

**Output**

As you can probably guess, the Output command is the reverse of the Input

command. You can use the Output command to send a single byte of data to a

port. Note that certain ports can cause the system to hang (especially those

dealing with system interrupts and the keyboard), so be careful with what you

send where!

-o 3fc 1

-

Port 3FCh is the modem control register for the first asynchronous port. Send-

ing a 01h to this port turns on the DTR (Data Terminal Ready) bit. A 00h will

turn all the bits off. If you have a modem which indicates this bit, you can

watch the light flash as you turn the bit on and off.

EXECUTION COMMANDS

**Go**

The Go command is used to start program execution. A very versatile

command, it can be used to start the execution at any point in the program, and

optionally stop at any of ten points (breakpoints) in the program. If no

breakpoints are set (or the breakpoints are not executed), program execution

continues until termination, in which case the message "Program terminated

normally" is sent. If a breakpoint is executed, program execution stops, the

current registers are displayed, and the DEBUG prompt is displayed. Any of the

DEBUG commands can be executed, including the Go command to continue execution.

Note that the Go command CANNOT be terminated by Cntl-break. This is one of

the few commands which cannot be interrupted while executing.

-g =100

The Go command without breakpoints starts program execution at the address (in

this case CS:0100) in the command. The equal sign before the address is

required. (Without the equal sign, the address is taken as a breakpoint.) If

no starting address is specified, program execution starts at CS:IP. In this

case, since no breakpoints are specified, CLOCK.COM will continue execution

until the cntl-break key is pressed and the program terminates. At this time,

you will get the message "Program terminated normally". Note that, after the

termination message, the program should be reloaded before being executed.

Also, any memory alterations (storing data, etc.) will not be restored unless

the program is reloaded.

-g 276 47c 528 347

This version of the control command will start the program and set breakpoints

at CS:276, CS:47C, CS:528 and CS:347. These correspond to locations in

CLOCK.COM after the screen is cleared, and the day, date and time are dis-

played, respectively. The program will stop at whichever breakpoint it hits

first. Note that the second and third breakpoints will only be displayed at

two times - when the program is started and at midnight. If you care to stay

up (or just change the time in the computer), and set a breakpoint at 47C, t

will stop when the program is started, and again at midnight.

Some notes about breakpoints. The execution stops just before the instruc-

tion is executed. Setting a breakpoint at the current instruction address will

not execute any instructions. DEBUG will set the breakpoint first, then try to

execute the instruction, causing another breakpoint. Also, the breakpoints use

Interrupt 3 to stop execution. DEBUG intercepts interrupt 3 to stop the pro-

gram execution and display the registers. Finally, breakpoints are not saved

between Go commands. Any breakpoints you want will be have to be set with each

Go command.

**Trace**

Along the same lines as Go is the Trace command. The difference is that,

while Go executes a whole block of code at one time, the Trace command executes

instructions one at a time, displaying the registers after each instruction.

Like the Go instruction, execution can be started at any address. The start

address again must be preceeded by an equal sign. However, the Trace command

also has a parameter to indicate how many instructions are to be executed.

-t =100 5

This Trace command will start at CS:100 and execute five instructions. Without

the address, execution will start at the current CS:IP value and continue for

five instructions. T alone will execute one instruction.

When using Trace to follow a program, it is best to go around calls to DOS

and interrupts, as some of the routines involved can be lengthy. Also, DOS

cannot be Traced, and doing so has a tendency to hang the system. Therefore,

Trace to the call or interrupt and Go to the next address after the call or

interrupt.

ARITHMETIC COMMANDS

**Hexarithmetic**

The Hexarithmetic command is handy for adding and subtracting hexadecimal

numbers. It has just two parameters - the two numbers to be added and subtrac-

ted. DEBUG's response is the sum and difference of the numbers. The numbers

can be one to four hexadecimal digits long. The addition and subtraction are

unsigned, and no carry or borrow is shown beyond the fourth (high order) digit.

-h 5 6

000B FFFF

-h 5678 1234

68AC 4444

-

In the first example, we are adding 0005 and 0006. The sum is 000B, the

difference is -1. However, since there is no carry, we get FFFF. In the

second example, the sum of 5678 and 1234 is 68AC, and the difference is 4444.

WRAPUP

If you give it a chance, DEBUG can be a very useful tool for the IBMPC.

It is almost a requirement for debugging assembler language programs, as no

nice error messages are produced at run time. DEBUG does work at the base

machine level, so you need some experience to use it effectively, but with

practice, it will be your most useful assembler language debugging tool.

There yah have it!