FACULTAD DE INGENIERIA

Arquitectura de Computadoras

DEBUG y Assembler

Profesor: Fabio Bruschetti

Ver 2013-01

DEBUG/ASSEMBLY TUTORIAL

This is a debug tutorial to assist the student who needs to know the inner-workings of the Intel based computer. The objective of this material is to instruct the student in observing the contents of the microprocessor and all of the memory locations that the processor can address. After you become familiar with debug and how it looks at addresses, you will be introduced to machine level programming, using debug as an assembler. At the end of this tutorial you should have a good understanding of the IBM system board and low level programming. Debug is a small utility program that uses one letter commands followed by several parameters needed on the command line.

It may be helpful if you have debug up and running as you read this tutorial so that the commands discussed can be entered and tested as you proceed through the material. Change the directory to the directory that has Debug.com and enter the command "Debug". Debug will load into memory and display the debug prompt, which is the hyphen "-". When you see the hyphen, debug is waiting for you to enter any one of the one letter commands.

The following pages show all of the commands and the syntax used with each. A detailed explanation of each command will help you in the proper usage and meaning.

How to start debug

Debug can be started in one of two ways.

Method one - At the DOS prompt you enter:

```
DEBUG (return)
```

Debug will respond with the hyphen (-) prompt. When the prompt appears debug is waiting for you to enter one of its many one letter commands. Starting debug this way will allow you to work on the internal hardware of the computer and view the contents of all of the memory location in RAM. You can also load in as many as 128 sectors of a floppy or Hard disk and view, edit or move the contents to another location. (See L command)

At the end of this tutorial you will be able to do the following tasks by starting debug this way.

1. Look at the DOS data area in memory to determine what kind of equipment is installed on the mother board.
2. Look at the internal workings of DOS at such things as the keyboard buffer, real time clock, interrupt vector table, rom bios chip and the video ram chips to name a few.
3. Recover deleted files from a floppy or hard disk.
4. Recover data from a disk that is un-readable by DOS and is otherwise lost.
5. Do diagnostics on some of the hardware, such as, video display, printers, disk drives and chips.
6. Low level format a hard-drive
7. Finally, you can assemble small COM programs using the 8088 instruction set.

Note: Debug sets up a work area in memory of 65,535 (decimal) one byte locations which is equal to FFFF bytes in Hex. The first 256 (decimal) or 100 Hex bytes of this

area are set aside for what is called the Program Segment Prefix (PSP) of a program and must not be altered in any way. Whenever we load sectors or data in memory with debug, it must be put at a location starting at offset 100.

An example of a debug command is shown on the following line.

```
L 0100 0 0 80 (return)
```

In this command, We are telling debug to load into memory starting at offset 100, 80 (Hex) sectors from the A drive starting with sector 0. 80 Hex sectors is equal to 128 decimal sectors, so if each sector on the disk, stores 512 bytes then the total number of bytes loaded into memory is (512 X 128) or 65,540 bytes. (maximum). We will go into more detail in the following sections.

Method Two - At the DOS prompt you enter:

```
Debug \path\filespec (return)
```

Debug will then load itself into memory along with the file that is specified in the path\filespec field of the command line and put the first byte of the file at offset 100 of the work area. By starting debug this way, we are able to view, edit or move a COM program or an ASCII text file. This is a very convenient way to debug or fix a COM program that was written with debug at some other time.

Note: MS-DOS will allow only two types of programs to run under its control and they must end with the extensions of EXE or COM. The difference in these two program types is in the way DOS handles the maintenance portions of the program. This maintenance area, often called the Program Segment Prefix (PSP), is a 256 byte block of memory that must be set aside by the program and is needed by DOS to return control back to the operating system when the program terminates.

Without going into a lot of detail at this time, I will point out the major difference between these two types of programs.

COM Extensions

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

EXE Extensions

1. The EXE programs can be any size from (200 bytes-640k bytes)
2. The PSP must be setup by the programmer, when the program is assembled.
3. The programmer determines where the first instruction is in the program.
4. The EXE program uses separate segments for the data, code and stack area in memory.

From the above comparison you can see it is much more difficult to assemble an EXE program than it is a COM program. The debug utility program was designed to

work only with a COM program by setting up the PSP area each time we enter debug. Once in debug, we can start assembly of a program at offset 100 and not be concerned with PSP or where the data, code, and stack is located. It is possible to look at an EXE program with debug if we rename the program with a different extension before we load it into memory.

To avoid a lot of confusion when you work with the debug program, the 8088 CPU instruction set and assembly programming, refer to the following list showing the commands and instructions of each one.

It is important for you to understand that:

- Debug has a set of commands
- The microprocessor has an instruction set
- Assembly language programming has a set of mnemonic instructions and pseudo-operations.
- Debug commands A C D E F G H I L M N O P Q R S T U W ?
- 8088 Instructions

The microprocessor is a digital circuit that only responses to a series of one's and zero's. Data is input to the CPU on the data pins marked D0 - D7. This is equal to one byte or 8 bits. If the first byte of data that enters the processor is an instruction, the processor will respond by performing the proper operation. In other words, the CPU has a set of 8 bit or one byte codes that it response's to, that will cause an operation to take place inside the registers or on the buses of the system board. These codes are called instructions and all of the codes combined are referred to have the instruction set.

The following examples of several 8088 instructions will help you understand. These are real instructions that preform the specified operation.

```
Byte  D7----D0-----operation performed
   1. 10111000 ----moves the next two bytes into AX
   2. 00000101 ----LSB of the number (goes in al)
   3. 00000000 ----MSB of the number (goes in ah)
   4. 00000001 ----adds BX to AX- result goes in BX
   5. 10001001 ----moves the contents of BX to the
   6. 00011110 ----memory location pointed to by
   7. 00100000 ----the next two bytes.
   8. 00000001
```

There are 256 possible combinations of an 8 bit binary number that can be used as an instruction. It is virtually impossible for a human being to remember and work with all of these one's and zero's of the 8088 instruction set. It has become necessary for us to represent these binary numbers with the base 16 number system.(Hex) The Hex representation of binary does make it a little easier to work with. For example, the program listed above in machine code (1's and 0's) can be represented with the Hex digits for the instructions as shown below.

```
Byte  Hexa------operation performed
   1. B8 -------moves the next two bytes into AX
   2. 05 -------LSB of the number (goes in al)
   3. 00 -------MSB of the number (goes in ah)
   4. 01 -------adds BX to AX- result goes in BX
   5. 89 -------moves the contents of BX to the
   6. 1E -------memory location pointed to by
   7. 20 -------the next two bytes.
   8. 01
```

This kind of programming is called machine code because the instructions are entered using binary or Hex data which, is the real data the CPU is responding to. The complete set of machine instructions will not be listed here because it would serve no purpose at this time. If you find a need for this information you must refer to the INTEL 8088 technical manual.

Assembly language

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

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

The mnemonic form of that program is shown below.

```
mnemonic form----------machine code
MOV AX,0005------------B80500
ADD BX,AX--------------01C3
MOV [120],AX-----------891E2001
```

Debug Command Description

A- Assembles 8086/8088 mnemonics.

C- Compares two areas of memory.

D- Displays the contents of memory locations.

E- Enters data into specified memory locations.

F- Fills memory locations with specified values.

G- Runs an executable program from within debug.

H- Performs Hexadecimal math on two Hex numbers.

i- reads one byte of data from the specified port.

L- loads the contents of a disk file or sectors into memory.

M- moves or copies the contents of memory locations.

N- Names a file for the L or w command. (load or write)

O- outputs one byte of data to the specified port.

P- Executes a loop, a repeated string instruction, a software interrupt, or a subroutine.

Q- Quits a debug session.

R- Displays or modifies the contents of the cpu registers.

S- Searches the contents of memory for the specified string of data.

T- Executes one instruction at a time and dumps the contents of all the registers to the screen, then displays the next instruction to be executed.

U- Disassembles machine code into the mnemonic form of the program.

W- Writes the data in memory to the disk. The name given the file is the name entered with the n (name) command. The file size is given by setting the BX:CX register to the number of bytes to be saved.

?- Display a help screen on some versions of debug.

A (assemble)

Assembles 8088/8086 mnemonics into memory

This command is used to create executable machine code from assembly-language statements. All numeric values are in hexadecimal form and can only have from 1 to 4 digits.

SYNTAX - `A [address]`

PARAMETER

`address` - the location where you will store the assembly-language instruction. The address must be in Hexadecimal form and within the range of 0100-FFFF. If you don't specify an address, debug will default to the last address used.

An assembly-language instruction is made up of a mnemonic instruction word followed by none/one or two operands. The format of an instruction is shown below.

- MNEMONIC 1st parameter, 2nd parameter
- (destination) (source)
- op-code 1st operand, 2nd operand

```
MOV AX, BX
```

The op-code part of the instruction (MOV) tells the processor what to do. The 1st parameter (AX) is the 1st location of data that the operation will be performed on. The 2nd parameter (BX) is the second location of data that the operation will be performed on. These parameters are called operands. The 1st operand is the destination of the data, the second operand is the source of data. In other words data

moves from the second operand to the first operand. The above move instruction above will move the contents of the BX register into the AX register.

More examples:

`MOV AX,45` load the hex number 45 into the AX register

`MOV AX,[120]` load the contents of location 120 into the AX register

`ADD AX,[500]` add contents of location 500 to AX ,save the results into AX

Note: You must start assembly of a COM program at offset 100.

C (compare)

Compares two blocks of memory.

SYNTAX `C range address`

PARAMETERS

`range` - specifies the starting and ending addresses or the start address and length, of the first block of memory to compare. address - specifies the starting address of the second block of memory you want to compare. Example: C 100 220 500 compares the block of memory at offset 100 - 220 with the block starting at offset 500. C 100 L 120 500 compares the block of memory at offset 100 for a length of 120 to the block of memory starting at 500.

D (dump)

Displays or dumps the contents of memory onto the display.

SYNTAX `D [range]`

PARAMETER

`range` - specifies the starting and ending address or the starting address and the length, of memory to be displayed. If you don't specify a range debug will display 128 bytes starting at the end of the last address dumped.

Note: When you use the D command, debug displays memory contents in two portions: a hexadecimal (each byte value is shown in hex form) and an ASCII portion (each byte shown as an ASCII character).

```
 seg:offset-Hexadecimal portion --------------------------- ASCII
1A00:E000  42 57 00 75 4A 80 3E A8-56 08 74 43 80 3E A8 56  BW.uJ.>.V.tC.>.V
1A00:E010  20 74 3C 80 3E A8 56 30-74 35 F6 45 01 01 74 2F  t<.>.V0t5.E..t/
1A00:E020  A1 44 57 8B D8 98 3B C3-75 25 8A 1D FE 0D 80 4D  .DW...;.u%.....M
1A00:E030  01 02 EB 1D 8A 45 01 24-01 80 3E 42 57 00 74 04  .....E.$..>BW.t.
1A00:E040  0C A8 EB 06 0A 06 A8 56-0C 04 88 45 01 FE 0D 8A  .......V...E....
1A00:E050  1D 32 FF 03 DF 43 A1 44-57 89 07 FE 05 F6 45 01  .2...C.DW.....E.
1A00:E060  01 74 02 FE 05 E9 30 01-80 3E 43 57 00 74 1D A0  .t....0..>CW.t..
1A00:E070  A7 56 A8 10 74 03 E9 4F-01 24 07 08 45 02 80 65  .V..t..O.$..E..e
```

Example:

```
D 100 10F
```

Will display the following,

```
0A1F:0100 - 41 42 43 44 45 46 47 48-1B 24 00 00 00 00 00 00 ABCDEFG......
```

E (enter)

Enters data into the memory location specified. Data can be entered in hexadecimal or ASCII format. Data at the specified location will be lost and replaced with the new data.

SYNTAX `E address [list]`

PARAMETER

`address` - specifies the first location in memory for the entered data.

`list` - specifies the data that you want to enter into successive bytes of memory.

Using the address parameter

If you specify a value for the address without specifying the LIST parameter, debug displays the address and its contents and waits for you to enter you new data. At this point you can do one of the following.

1. Replace the byte value with a new value in hex. If the value you enter is not valid hexadecimal, debug will inform you of that.
2. Advance to the next byte in memory. To do this you press the space bar, and to change the value at the new location, you just type a new value. If you press the space bar more than eight times, debug starts a new line.
3. Return to the preceding byte. To do this you press the hyphen key. Each time you press the hyphen key, debug will go back in memory one byte.
4. Exit from the E command. To do this you just press the enter key.

Using the LIST parameter

If you specify values for the list parameter, the old value will be replaced with the new value. LIST values can be hexadecimal or string values. To enter string values, just enclose the string within single or double quotes.

Example: If you type,

```
E 100
```

debug displays the contents of offset 100

```
0A1F:0100 41._
```

To change this value you just type in the new value,

```
0A1F:0100 41.35
```

In this example, the value 41 at offset 100 was changed to 35.

You can enter consecutive byte values with one E command. Instead of pressing the enter key after you change a value, just press the space bar and debug will display the next location in memory, at which time you can enter a new value. Repeat this process as many times as you need to.

If you enter,

```
E 100
```

debug returns the value at offset 100

```
0A1F:0100 35._
```

if you press the space bar three times then debug shows the next three offsets.

```
0A1F:0100 35._ 42._ 43._ 44._
```

To change the byte at offset 100 you can hit the hyphen key three times and debug will go back to offset 100.

```
0A1F:0102 43._
0A1F:0101 42._
0A1F:0100 35._
```

Press enter to get out of the E command.

To enter data into a location using a String of text, enclose the string in quotes.

```
E 100 "This is a string of text."
```

This command will enter the string of ASCII characters into the 25 memory locations starting at offset 100.

(one location for each character and spaces)

F (fill)

Fills the addresses in the specified memory locations with the value you specify. You can specify the data in hexadecimal or ASCII values.

SYNTAX `F range list`

PARAMETERS

`range` - Specifies the starting and ending address, or the starting address and the length of the memory area you want to fill.

`list` - Specifies the data you want to fill with. Data can be hexadecimal or ASCII form. The ASCII form is a string of text enclosed in quotes.

Using the range parameter

If range contains more bytes than the number of values in LIST, debug assigns the values in LIST repeatedly until all bytes in range are filled. If any of the memory in the range specified is bad or doesn't exist, debug will return an error message to that affect.

Using the list parameter

If list contains more values than the number of bytes specified in range, debug will ignore the extra values.

G (go)

Executes the program that is in memory. The go command uses the IP register as a pointer to the next instruction that will be executed.

SYNTAX `G [=address] [breakpoint]`

PARAMETER

`address` - Specifies the address that program execution will begin at.

If you do not specify an address, debug executes the instruction at the address in the CS:IP registers. If the program ends with an INT 20 instruction, the IP register is reset back to offset 100. If the program ends with an INT 3 instruction, the IP register will remain pointing to the next instruction after the last instruction executed.

`breakpoint` - Specifies from 1 to 10 breakpoints that can be entered with the Go command.

Using the address parameter

You must precede the address with an equal sign (=) to distinguish the starting address from the breakpoint address.

Specifying breakpoint

The program stops at the first breakpoint that it encounters and dumps the contents of all registers, the status of the FLAGS and displays the last instruction that was executed.

Examples: If you type,

```
G cs:200
```

debug will execute the program up to offset 200, then dump the contents of all the registers on the screen.

If you type,

```
G=cs:200
```

Debug will execute the program starting at offset 200 to the end of the program.

If you type,

```
G
```

Debug will use the CS:IP register to get the address for the next instruction to be executed. You should always look at the IP register before you enter g, to make sure that it is pointing to the next instruction that you want to execute. To look at the IP register, just enter rip (return), debug will display the current value in IP and prompt you for a new value if you desire. The IP register should always be at 0100 to run the entire program.

Be careful when ending the program with an INT 3 because all registers are preserved and are not reset to their initial values, IP will be pointing to the next address after the end of the program and not your program.

INT 20 will reset all the registers to zero and the IP to 0100 when the program ends.

See P and T commands

H (Hex)

Performs hexadecimal math on the two parameters you supply.

SYNTAX `h value1 value2`

PARAMETER

`value1` - represents any hex number between the range of 0000 - FFFFh.

`value2` - represents any hex number between the range of 0000 - FFFFh.

Debug first adds the two parameters you specify and then subtracts the second from the first. The results are displayed on one line. First the sum, then the difference.

If you type,

```
h FF 2C5
```

debug does the calculations and displays,

```
03C4 FE3A
```

03C4 is the sum and FE3A is the difference.

I (input)

inputs and displays a byte from the specified port

SYNTAX `I port`

PARAMETER

port - specifies the port address. The port address can be an 8 or a 16 bit value. It is possible to have up to 65,535 port addresses.(FFFF). See port

If you type

```
i 3bc
```

debug will display 1 byte of data from the parallel port.

L (Load)

Loads a file or the contents of the specified sector from the disk into memory.

SYNTAX `L address`

```
L address drive start number
```

PARAMETER

`address` - Specifies the memory location where you want to load the file or sector contents. If you don't specify an address, debug will use the address in the CS:IP registers

`drive` - specifies the drive that contains the disk that specific sectors are to be read. This is a number from 0 - 4.

0 = A

1 = B

2 = C

3 = D

`start` - specifies the hexadecimal number of the first sector whose contents you want to read.

`number` - specifies the hexadecimal number of sectors you want to load.

Using L with no parameters.

If you enter L without parameters, debug will re-load the file that was specified on the DEBUG command line into memory at offset CS:0100. Debug also sets the BX:CX registers to the number of bytes for the file size. If you didn't enter a file name on the command line debug will load the file that was specified by the n command.

Note: The name of the file debug is using will be stored at offset 0080. To see the current name, use the d command to dump offset 80. Enter d 80 and debug displays the name of the current file. You should get in the habit of doing this so the file or program you are working on will be sure to get saved to the disk.

Using L with the address parameter

If you use L with just an address parameter, debug will load the data at the specified offset in memory.

Using L with all parameters

If you use all parameters, debug will load the specified sectors into the specified offset.

Loading an EXE program

Debug ignores the address parameter for EXE files. If you specify an EXE file, debug relocates the file to the loading address specified in the header of the EXE file. The header itself is stripped off the EXE file before it is loaded into memory, so the file size on the disk will be different than the file size in memory. If you want to examine an EXE file, rename it with a different extension.

```
REN FILE1.EXE FILE1.TXT
```

Example: Suppose you start debug and at the prompt you type,

```
-n Filename.com
```

You can then enter L and debug will load the file with the name you specified from the disk.

Suppose you want to load the root directory from the disk in the A drive. The root directory starts at sector number 5 and is a total of 7 sectors in size.

You enter,

```
L 0100 0 5 7
```

debug loads in the 7 sector starting at sector 5 from the disk in the A drive and stores it at offset 0100.

M (move)

Copies the contents of a block of memory to another block of memory.

SYNTAX `m range address`

PARAMETER

`range` - specifies the starting and ending addresses, or the start and length of the memory area whose contents are to be copied.

`address` - specifies the starting address of the location where you want to copy the contents of range.

Example: If you type the following command,

```
m 0100 110 500
```

Debug copies the block of data that starts at offset 100 and ends at offset 110, to offset 500 in memory.

Note: You can use this command if you have to insert an instruction in the program that has already been written. Suppose you have to insert the instruction Mov ax,5600 in the program you are working on at offset 160. Enter,

```
m 160 FFFF 1000
```

Then assemble the instruction at offset 160.

```
A 160
```

```
CS:0160 MOV AX,5600 (return)
```

To get the rest of the program back together (it was moved to offset 1000). Enter,

`m 1000 FFFF 163` (163 is the next offset after MOV AX,5600)

The program should all be in order now.

N (name)

Specifies the name of a file that will be used by the L or the W command.

SYNTAX `n [drive:][path]\filename`

PARAMETER

`[drive:][path]filename` - specifies the location and name of the file you want to work with.

Example:

Suppose you started debug with no filename on the command line, at the debug prompt you just enter the filename you want.

```
-n break.com
```

Then enter L and the file is loaded into memory.

Suppose you are working on a program and you want to work on a different program, just enter,

```
-n filename.ext
```

```
L (return)
```

and the new file is loaded into memory.

O (output)

Sends one byte to the specified port.

SYNTAX `o port byte`

PARAMETER

`port` - specifies the output port address. Can be an 8 or a 16 bit port.

`Byte` - specifies the hexadecimal that will be sent to the specified port.

Example: To send the letter A to the parallel printer, Just enter,

```
o 3bc 41
```

See port

P (proceed)

Executes a loop, a repeated string instruction, a software interrupt, or a subroutine, or traces through any other instruction.

SYNTAX `p [=address] [number]`

PARAMETER

`address` - specifies the address of the first instruction to be executed. If you don't specify an address, debug will use the address in the CS:IP registers. (see G command)

`number` - specifies the number of instruction to execute before returning control to debug. The default is one.

Note: p is similar to g or the t command and is used for debugging a program by single stepping through the program. The p command however, will execute loops and INT instructions. The p command cannot be used to trace through the ROM bios program.

Q (quit)

Exits debug and returns control to DOS.

SYNTAX `Q`

PARAMETERS none

Note: Be sure to write your program or data to the disk before you enter Q, otherwise it will be lost.

R (register)

Displays or changes the contents of one or more of the micro-processor registers.

SYNTAX `r [register-name]`

`r` by itself will display all of the registers

`rf` displays the flag register

PARAMETER

`register-name` - specifies the name of the register you want to display.

Using the r command

If you specify a register name, debug displays the 16 bit value of that register in hexadecimal form followed by the colon. If you want to change the value of the register, type the new value plus enter. If you don't want to change the value just press the enter key.

Valid register names

The following are valid register names:

- AX BX CX DX
- SP BP SI DI
- ES SS DS CS
- IP F

Using the F character (FLAG)

If you use the F character instead of a register name debug displays the current status of the flags register. Each flag has a two letter code to shown the condition of the flags. To set or clear the flags use the following list of two letter codes.

```
FLAG NAME------------SET--------------CLEAR
Overflow-------------ov---------------nv
Direction------------dn---------------up (increment)
Interrupt------------ei (enabled)-----di (disabled)
```

```
Sign-----------------ng (neg)---------pl (positive)
Zero-----------------zr---------------nz
Auxiliary carry------ac---------------na
Parity---------------pe (even)--------po (odd)
Carry----------------cy---------------nc
```

Default settings for debug

When you start debug, the segments registers are set to what ever segment is free in your computer at the lowest possible memory location, the IP register is set to 0100, all flags are cleared and the remaining registers are cleared to zero except the SP register, which is set to FFFE.

Example:

To put the number 1234 into the AX register, enter

```
RAX (return)
```

you will see

```
AX 0000:
```

after the colon enter 1234 (return)

To see just the flags register enter,

```
rf (return)
```

you will see

```
NV UP DI NG NZ AC PE NC- _
```

at the hyphen enter the two letter code for the flag, you want to affect.

```
pl ei cy (return)
```

S (search)

Searches a range of addresses for a pattern of one or more bytes.

SYNTAX `s range list`

PARAMETER

`range` - specifies the start and end address of the range you want to search.

`list` - specifies the pattern of bytes or a string that you want to search for. Separate each byte value with a comma or a space and enclose the string in quotations marks.

Example: To search a block of memory for the pattern of bytes, 41 55 66 located at offset 100 thru 250. Enter,

```
s 100 200 41 55 66 (return)
```

debug will display the address of all locations that contain this pattern of bytes.

To search the same block of memory for the string, Data Institute, Enter

```
s 100 250 "Data Institute"
```

debug will return all of the locations that contain that string.

Note: You must have the exact bytes or the proper case letters for the search command. It is case sensitive.

T (trace)

Executes one instruction at a time and displays the contents of all of the registers, the condition of the flags and the next instruction to be executed.

SYNTAX t `[=address] [number]`

PARAMETER

`address` - specifies the start address to start tracing. If you don't specify the start, debug will use the CS:IP registers for the next instruction.

`number` - specifies the number of instructions to be executed. Default number is one.

Note: t is like the g and p command except it cannot go through a sub-routine or an INT instruction. I Would recommend using the p command for all single stepping and debugging. However, t can be used to trace through the ROM bios program. If you want to unassemble the ROM bios program use t.

U (unassemble)

Disassembles bytes and displays their source code listing in mnemonic form with their addresses and values.

SYNTAX `u [range]`

`u` (by itself u will unassemble 20 bytes)

PARAMETER

`range` - specifies the starting and ending addresses or the starting address and the length, of code you want to unassemble.

Example: To unassemble 16 bytes (10hex) starting at 0100, enter,

```
u 0100 L 10
```

debug displays,

```
CS:0100 MOV AX,5600
CS:0102 MOV DX,200
CS:0105 INT 21
CS:0107 MOV AH,7
CS:010B CMP AL,1B
CS:010D JZ 0111
CS:010F JMP 0107
```

You could have also entered,

```
u 100 110
```

and debug would display the same information.

W (write)

Writes a file or a specific number of sectors to the specified disk.

SYNTAX `w address drive start number` (to write sectors to the disk)

`w address` (to write files to the disk)

PARAMETER

`address` - specifies the beginning memory address of the file you want to write to the disk file. If you don't specify, debug will start at CS:0100.

`drive` - specifies the drive that will be the destination disk.

- 0 = A
- 1 = B
- 2 = C
- 3 = D

`start` - specifies the start sector number in hexadecimal that you want to write to.

number - specifies the total number of sectors you want to write.

Saving a file to the disk

NAME

To write a file to the disk you must first, use the name command (n) to name the file. You must specify the path in the name command such as A:\advanced\break.com.

SIZE

After the file is named, you must specify the size of the file in total bytes, by setting the BX:CX registers to the number of bytes. Be careful that you have the BX set to zero before you write a COM file to the disk. Each digit in BX represents 65,000 decimal bytes. For our purposes BX will be always set to 0.

Key Words

Assembly Language - is a very low level programming language that uses a mnemonic prefix followed by one or two operands. The statements are in plain English using common words. For example,

MOV AX,34

MOV [120],AX

SUB AX,BX

They are almost understandable because of the mnemonic words used in the instructions. In the first line, the instruction MOV AX,34 simply means, Move the number 34 into the AX register. In the second line, MOV [120],AX, means move the contents of the AX register into the memory location at offset 120. The last line, SUB AX,BX, means subtract the contents of the BX register from the contents of the AX register and put the results in the AX register.

Note: Debug will allow us to write small programs using the mnemonic form of the instruction set for the 8088 micro-processor. We will be required to specify the address that is used by the program in the form of an offset number. This means we have to pay attention to the locations in memory of all of the instructions that the program uses.

Bios call

When a program or device executes an interrupt sub-routine inside the ROM BIOS chip. (see INT)

Bit

This is the smallest unit of digital data. One bit represents either a one or a zero in binary logic.

Byte

One byte is 8 bits arranged in order with the least significant bit to the right. The bits are usually identified by the names, D7 D6 D5 D4 D3 D2 D1 D0 in that order.

Code

Is what we call the data in memory. Machine code refers to the program instructions in hexadecimal form. Assembly code refers to the program code in assembly language form. Text code refers to the ASCII code that represents the messages of the program.

Data

Is the binary information that is in the memory chips of the computer and is in the form of one byte or 8 bits. Debug displays the binary data in the form of Hex numbers so, if you want to see what it looks like in binary (the way it is) you will have to convert the Hex number to binary bits. Sometimes we represent data using two bytes which is called a word or four bytes which is called a double word.

Debug

A program used to view, edit, and save data in memory and on the disk. Can also be used to assemble small COM programs. The word also applies to the fixing of computer programs. As the story goes, a computer went down and required some repairs. After a lengthily time the problem turned out to be caused by a bug that got caught in one of the relays.

Display

Another word for the monitor that is used to display the output of a computer program. Common displays in use today are the Mono-graphics (MGA), Color graphics (CGA), Video graphics (VGA) and Super Video graphics (SVGA)

Double word

In the 8088/80286/80386 a double word is two words or four bytes. Double words are stored in reverse order in memory with the most significant byte in the highest offset location.

DOS call

When a program or a device executes an interrupt sub-routine into IBMDOS.COM. All DOS calls use the INT 21 instruction. (see INT)

Echo

When a character is echoed to the display it simply pops up on the screen. Every time we press a key on the keyboard the character is echoed to the screen.

Flag

The processor has an 11 bit register called the flag register that saves the results of the arithmetic operations. (see the r command) LOOP - An 8088 instruction that is used for repeated operations. The Syntax is LOOP xxxx, where xxxx is the address of the instruction that will be executed. The CX register must be set with the loop count, every time the loop instruction is executed, CX will decrement by one until it

reaches zero, at which time the loop instruction is over and the next instruction is executed.

Int

An instruction that is used in a program to call a software interrupt sub-routine that may be located in the ROM BIOS chip or the MS-DOS operating system. INT 10, for example will call an interrupt 10 Sub-routine that controls the video on the display.

Interrupt

Refers to interrupting the cpu for a period of time. There are two kinds of interrupts you should know. A software interrupt is an interrupt that occurs when an INT instruction is executed from within a program. A hardware interrupt occurs when a piece of hardware produces an interrupt signal on the interrupt controller chip.

Port

Is an interface card that plugs into one of the slots on the mother-board. Each card has its own I/O (input/output) address. The 8088-80486 processors can have up to 65,535 ports. Ports are accessed with the OUT and the IN instruction. OUT 03BC 41 is an example of how data (41 HEX or upper case A) is sent to the port with an address of 03BC. Some common ports for the IBM pc are listed below.

Printer Ports: Name Address

Parallel: LPT1 03BC	Displays: Mono	Floppy
LPT2 0378	03B0-03BF	03F0-03F7
LPT3 0278	Displays: Color	Other: Game
Serial: COM1 03F8	03D0-03DF	0200-020F
COM2 02F8	Disk Drives: Hard	
COM3 03E8	0320-032F	
COM4 02E8		

Offset

Refers to the address of a memory cell inside the ram chips. When we assembly or load a COM program it will always be located in some segment of memory. The very first memory location of that segment is at offset 0 and the last memory location is at offset FFFF. A COM program must be in the same segment of memory, that means a COM program is smaller than 65,535 bytes.(see segment)

Register

A temporary storage location that is made up of a latch of some kind. The 8088 micro-processor has fourteen 16 bit registers to process the data in a computer.

Four are for data - AX,BX,CX,DX

Four are for segment addresses - ES,DS,SS,CS

Four are for index addressing - SP,BP,SI,DI

One is the instruction pointer - IP

One is the flag register - Flag

Reset

A pin on the 8088 - 80486 that is used to force the processor into a reset condition. (boot) Reset is also used to describe the act of clearing a bit or setting it to zero.

Reset Address

Is a pre-set address that all processors will point to each time the reset pin is active. The reset address is F000:FFF0 or absolute physical address of FFFF0. There is a jump instruction at this location (ROM BIOS CHIP) that will point to the beginning of the ROM BIOS program.

Stack

Is a location in memory that is used by a program to store data for a short period of time. The stack is located at offset FFFE in the top of memory in a com program. The PUSH and the POP instruction use the stack to store the contents of the registers when an INT instruction is executed.

String

Is a group of bytes used to represent information. For example,

" This is an example of a string ".

Sub-routine

Refers to a group of instructions that performs a specific task and is located away from the main program. To use the sub-routine a CALL instruction is executed in the main program, when all the instructions in the sub-routine are finished, a RET instruction will return control to the calling program.

Word

In the 8088/80286/80386 a word is two bytes and is stored in reverse order with the most significant byte in the highest offset location.

The 8086/8088 Micro-processor

All information that follows applies to the 8086 family of micro-processors, that includes the 80286, 80386, and the 80486. However, we will discuss only the instruction set of the 8088 cpu. If working with one of the other micro-processors, you must refer to the additional instruction set for that CPU. These processors are upward compatible which means, all of the processors will run the 8088 programs.

Intel's 8088 8 bit micro-processor

The 8088 micro-processor is a 40 pin HMOS TTL type chip utilizing a single 5 volt power supply source. IBM calls it a 16 bit processor, but in reality it is only an 8 bit processor. This discrepancy comes from the fact that it only has 8 data pins on the outside, but it has 16 bit data registers on the inside. In other words, the data is actually transferred on the system board 8 bits at a time but the data is operated on 16 bits at a time inside the 8088 registers. There are several registers inside the

8088 that are used to hold data and or to modify it on a temporary basis. These registers can be thought of as 16 bit d latches or 16 bit JK flip-flops. After an operation is preformed on the data in one of the registers, it must be moved to a location in ram, for a more permanent type of storage until it is saved to a disk file.

8088 8-BIT HMOS microprocessor

- 8 bit data bus interface
- 8 - 16 bit internal registers
- direct addressing of 1 meg memory
- 14 working registers (16 bits)
- byte, word and block operations
- 2 clock rates (4.77 MHz and 8 MHz)

Pin Function Description

SYMBOL PIN # TYPE NAME AND FUNCTION

AD7-AD0:9-16 I/O ADDRESS DATA BUS: These pins are the time multiplexed memory/IO address and data lines.

A15-A8:2-8,39 O ADDRESS BUS: These pins provide address bits 8 through 15 for the entire bus cycle.

A19/S6:5-38 O ADDRESS/ STATUS: These pins are the

A18/S5 4 most significant address lines for

A17/S4 memory operations.

A16/S3

RD: 32 O RESD: Read strobe indicates that the processor is performing a memory or IO read cycle, depending on the state of the IO/M pin or S2.

READY: 22 I READY: is the acknowledgement from the addressed memory or I/O device that it will complete the data transfer. The RDY signal from I/O or memory is synchronized by the 8284 clock generator to form READY.

INTR: 18 I INTERRUPT REQUEST: is a level triggered input which is sampled during the last clock cycle of each instruction to determine if the should enter into an interrupt acknowledge operation. A subroutine is vectored to via an interrupt vector look-up table located in system memory. It can be internally masked by software re-setting the interrupt enable bit.

TEST: 23 I TEST: input is examined by the "wait for test" instruction. If the TEST input is low, execution continues, otherwise the processor waits in an "idle" state.

NMI 7 I NON-MASKABLE INTERRUPT: is an edge triggered input which causes a type 2 interrupt. A subroutine is vectored to via an interrupt vector look-up table located in system memory. This in-put is not maskable by software.

RESET 21 I RESET: causes the processor to terminate its present activity. Must be active high for at least four clock cycles.

CLK: 19 I CLOCK: provides the basic timing for the processor and bus controller. It is asymmetric with a 33% duty cycle.

VCC: 40 VCC: is the +5 V +/- 10% power supply pin.

GND 1,20 GND: are ground pins.

MN/MX: 33 I MINIMUM/MAXIMUM: indicates what mode the processor is in.

Memory organization

The 8088 provides a 20 bit address to memory which locates the byte being selected. This memory is arranged as a block of up to 1 megabyte, addressed as 00000(H) to FFFFF(H). This memory is further divided into smaller blocks called segments, that are 64K bytes per segment. There are four segment types called CODE, DATA, EXTRA and STACK that are used for the data and code of a program. Addressing memory is accomplished through the use of one of four high speed segment registers called CS, DS, ES, and SS in combination with an IP register (instruction pointer). The segment register points to the segment in memory being used and the IP register points to an offset within that segment. See Fig. X & X

The registers are divided into groups that is based on what they are used for. General purpose registers are 16 bits or 8 bits

The 8088 instruction set (41 instructions)

Arithmetic instructions

Data transfer instructions		Arithmetic instructions	
MOV	move	ADD	addition
PUSH, POP	stack operation	INC	increment
XCHG	exchange	SUB	subtract
IN,OUT	input/output	DEC	decrement
		NEG	negate (two's comp)
		CMP	compare
		MUL	multiply
		DIV	divide

Logical instructions		String instructions	
MOV	move	ADD	addition
NOT	complement	MOVS	move string
AND	and	CMPS	compare string
OR	inclusive or	SCAS	scan string
XOR	exclusive or	LODS	load from a string
TEST	test bits	STOS	store into string
SHL,SHR	shift left/right		

ROL,ROR	rotate left/right		

Transfer of control instructions	
CALL	goto a sub-routine
RET	return from a sub-routine
JMP	jump
JZ,JNZ	conditional jumps
LOOP	iteration
LOOPNE	conditional iteration
INT	interrupt
IRET	return from interrupt

Processor Control	
CLC,STC	clear/set flags
HLT	halt CPU

How to use the Data transfer instructions (MOV)

There are six different data-addressing modes used for data transfers with the 8088 CPU. A number enclosed in brackets is a memory location at the offset specified by the number. [150] is a memory cell at offset 150 in RAM that can hold 8 bits or one byte of data. If a register is enclosed with brackets [BX], then the CPU will go to the memory location specified by the register for the data.

```
MOV AX,BX
```

register, register Data is transferred from the BX reg. into the AX reg. (16 bit transfer)

```
MOV AH,BL
```

register, register Data is transferred from the BL reg. into the AH reg. (8 bit transfer)

```
MOV AX,5
```

immediate The hex value in the source (5) is moved directly into the AX reg.(16 bit transfer) 5 is really 0005

```
MOV AH,5
```

immediate The hex value in the source (5) is moved directly into the AH reg. (8 bit transfer) 5 is really 05

```
MOV AX,[200]
```

direct The brackets are used to enclose a memory location. The value stored at offset location 200 and 201 is moved into the into the AX reg. Each memory location is one byte and since this a 16 bit or two byte transfer it will use two memory locations. The Al reg. will get the value at 200 and the AH will get the value at 201.

```
MOV AL,[450]
```

direct The value stored at offset450 is moved into the AL reg. Since this is an 8 bit transfer it will only use one memory location.

```
MOV AX,[BX]
```

indirect The value stored at the location that the BX reg points to will be transferred into the AX reg.Suppose the BX reg has the value 345 in it. The AX reg would than get the value at location 345 and 346. (16 bits)

```
MOV CL,[BX]
```

indirect same as above but is an 8 bit transfer

```
MOV AX,[BX+5]
```

base+displacement Same as above but the number is added to BX to get the location of the data.(16 bit transfer)

```
MOV AH,[BX+5]
```

base+displacement Same as above but 8 bit transfer.

```
MOV AX,[BX+SI+8]
```

base+index+8 Here there are two reg. displacement used + a value. The CPU will calculate the location in memory using the values in the BX and SI reg. plus the value specified.(16 bit transfer)

```
MOV AL,[BX+SI+6]
```

base+index+6 same as above but 8 bit displacement transfer.

Note: We are only going to assemble programs using modes 1,2,3 and 4

How to use the PUSH and POP transfer instructions

```
PUSH AX
```

```
POP AX
```

There are four general purpose registers that can be used by our program to transfer data and perform arithmetic operations. This may seem like enough, but when writing a program you may find yourself looking for a few more registers to work with. The PUSH and POP instruction will allow us to save the contents of the registers being used into the stack area of memory. Once the contents are saved, we can re-use them for more operations and when finished, restore them to their original values they had before they were saved.

PUSH CX will cause the contents of the CX register to be saved in the stack area of memory. The stack area is located at the top of the segment our program is in at the time. The first location available in the stack is located at offset FFFE and will decrement downward in memory. The stack is what we call "First In and Last Out" and we must pay attention to how we use the PUSH and POP instructions. Think of the stack as a pile of dishes, The first dish we put in the shack will have to be the last dish we take off of the stack. If we try to take the dishes off in any other order the pile will most likely crash. If we do the same thing in our program, the computer will crash.

Example: Look at the following program,

```
MOV CX,115F
MOV AX,BX
OUT DX,AL
INC CX
PUSH CX
PUSH AX
MOV DX,03BC
MOV AX,[120]
MOV CX,BX
OUT DX,AL
POP AX
POP CX
RET
```

The first 3 instructions are using the CX and AX registers to hold values needed in this part of the program. After the CX register is incremented, two push instructions are executed,.(PUSH CX and PUSH AX) that will save the current value of CX and AX on the stack. CX and AX can then be used for more instructions, then POP AX and POP CX in that order, will restore the values that were in them in the first part of the program.

When our program uses an INT instruction, the CPU will change the IP reg to point to a sub-routine program inside the ROM bios chip or the IBMDOS.SYS program in low memory. These sub-routines will use all of the general purpose registers that are program was using before the INT instruction was executed. IBM says all registers are preserved (saved) when an INT is executed except the AX register. I would suggest that you PUSH the AX register every time you use the INT instruction.

How to use Arithmetic instructions

All arithmetic must be done in one of the general purpose registers. To add two numbers, put one number in the a register and the other can be stored in another register or memory location. I suggest the AX register, because it was optimized to work as an accumulator in arithmetic operations. Below are three forms of this instruction that you should be familiar with.

16 bit ADD instructions. (any instruction that uses AX, BX, CX or DX)

```
ADD AX,BX
```

Value in BX is added to the value in AX and the result is placed in AX.

```
ADD AX,[120]
```

Value stored at offset 120 and 121 in memory is added to the value in AX and the result is placed in AX. The CPU will automatically calculate the number of memory locations to use when it fetches data from memory. In this case, the register specified in the instruction is a 16 bit register and requires two bytes of data from memory which starts at offset 120 (the least significant byte) and 121 (the most significant byte).

```
ADD AX,5
```

The hex number 5 is added to the value in AX and the result is placed in AX. In this example, the number 5 is actually 0005 in hex. When we assemble a program it is OK to leave out the leading zeros as I did here, Debug will just insert them for you.

8 bit ADD instructions. (any instruction that ---- uses AH, AL, BH, BL, CH, CL, DH, DL are 8 bit instructions)

```
ADD AH,AL
```

Value in AL is ADDED to the value in the AH and the result is placed in AH.

```
ADD AH,[150]
```

The value stored at offset location 150 in memory is added to the value in AH and the result is placed in AH.

Example: This short program will add to hex numbers and store the result in memory.

`MOV AX,3405`	put first number in a register
`MOV [200],AX`	mov the number to a memory location for temporary storage
`MOV AX,5892`	put the second number in a register
`ADD AX,[200]`	add a number in the register and put the result back in the register
`MOV [202],AX`	save the result at a memory location
`INT 20`	end the program

```
INC AX
```

Increment will increase the value in the specified register or memory location by 1 each time it is executed. It is the equivalent of ADD AX,1, where 1 is added to the specified register. INC can be used as many times as you like. In this example, the AX register will be increased by one each time it is executed. So if the AX register has the number 0005 in it, the INC AX instruction will increase it to 0006.

```
INC BL
```

Increments the value in the specified 8 bit register.

```
INC BYTE [150]
```

Increments the byte value at the specified memory location.

```
INC WORD [150]
```

Increments the word (16 bit number) at the specified memory location.

```
SUB AX,CX
```

The 16 bit value in the source register is subtracted from the value in the destination register and the result is placed in the destination register.

```
SUB CX,[160]
```

The value stored at the specified location is subtracted from the value in the specified register and the result is placed in the destination register.

```
SUB AL,AH
```

The 8 bit value in the source register is subtracted from the value in the destination register and the result is placed in the destination register.

Note: If the result of the subtract is zero, the zero flag is set.

```
DEC AX
```

Decrement will decrease the value in the specified register or memory location by 1 each time it is executed. It is the equivalent of SUB AX,1, where 1 is subtracted from the specified register. DEC can be used as many times as you like. In this example, the AX register will be decreased by one each time it is executed. So if the AX register has the number 0005 in it, the DEC AX instruction will decrease it to 0004.

```
DEC AL
```

Decrements the value in the specified 8 bit register.

```
DEC BYTE [250]
```

Decrements the one byte value stored at the offset specified. In this example the byte stored at offset 250 is decreased by one.

```
DEC WORD [200]
```

Decrements the word (16 bit number) stored at the offset specified. In this example the 16 bit number stored at offset 200 is decreased by one.

Note: If the result is zero, the zero flag is set.

```
NEG AX

NEG AL

NEG BYTE [200]
```

Negates the value in the specified register or memory location. Negate is the two's complement of a number.

```
CMP AX,56

CMP AL,BL

CMP AL,[128]

CMP AX,[150]
```

The compare instruction will compare the numbers in both registers or memory locations and if they are equal the zero flag is set. This instruction is used to change the flow of a program based on the condition of the zero flag. The following program will demonstrate how this is done.

`MOV AH,09`	Print the message at offset 200. Message ends with hex 24
`MOV DX,200`	hex 24 is the $ character)
`INT 21`	
`MOV AH,7`	Wait for a key to be pressed and put the ASCII code
`INT 21`	for it in the al register. Compare the ASCII code for
`CMP AL,1B`	the key that was pressed with 1B which is the ASCII code for the escape key. If the compare is equal, the zero flag is set.
`JNZ 100`	Here the program can go in one of two directions. If the zero flag is not zero, the instruction at offset 100 is executed.(JNZ or jump if not zero to) This causes the program to read a new key value and repeat. If the key pressed was the escape key, the zero flag is set and the next instruction is executed.
`INT 20`	End the program and return to DOS.

Note: Assemble this program beginning at offset 100 and use the P command to single step through the instructions. You must assemble the message at offset 200 and end it with the dollar sign.

```
A 200

xxxx:0200 DB "PRESS THE ESC KEY IF YOU ARE TIRED OF READING
THIS MESSAGE.$"

MUL AX,BX

MUL AL,05

DIV AX,CX

DIV AL,BL
```

MUL and DIV are special instructions and will not be explained here.

Special debug instructions

```
DB (define byte)

DW (define word)
```

`DB` and `DW` is used to define a data type in memory.

DB "This is a message" will store the ASCII string that is enclosed in quotations at the address of the assembly instruction.

For example, if you enter,

```
A150

XXXX:0150 DB "My name is John Smith"
```

then, the ASCII string My name is John Smith is stored in memory, starting at offset 150.

You can also use DB to load specific byte of data into memory.

```
DB 41 42 43 44 45
```

Would load ASCII letters A B C D E into memory at the specified memory location .

The first 3 instructions are using the CX and AX registers to hold values needed in this part of the program. After the CX register is incremented, two push instructions are executed,.(PUSH CX and PUSH AX) that will save the current value of CX and AX on the stack. CX and AX can then be used for more instructions, then POP AX and POP CX in that order, will restore the values that were in them in the first part of the program.

How to use Logical instructions

To predict the results of the logical instructions, you must convert the Hexadecimal number into binary numbers and then perform the logical function on each bit. Logical instructions are for bit manipulation of the registers specified. This is the only way we can set or clear the bits inside any of the registers.

The AND instruction is used to clear individual bits.

The OR instruction is used to set individual bits.

The XOR instruction is used to clear a register to zero.

USE the

Decimal	Hexadecimal	Binary
1	1	0001
2	2	0010
3	3	0011
4	4	0100
5	5	0101
6	6	0110
7	7	0111
8	8	1000
9	9	1001
10	A	1010
11	B	1011
12	C	1100
13	D	1101
14	E	1110
15	F	1111

Converting from Hex to Binary and Binary to Hex.

To convert HEX to binary, You just find the four bit binary number that represents each HEX digit.

```
      A5 (hex number)
    /   \
   /     \
1010     0101 (binary number)
```

To convert BINARY to HEX, You just find the hex digit value for each four bit group of binary digits.

```
10001101 (binary number)
\  /\  /
 \/  \/
  8   D (hex number)
```

If the AX register has the HEX number 45A1 in it. Then,

```
AX = 45A1 (hex number)
0100010110100001 (binary number)
| 4|| 5|| A|| 1|
```

How to predict the LOGICAL instruction

To predict the result of any logical instruction, You must convert all values into BINARY, then perform the logical function on each pair of bits separately. The truth tables for the logical functions are shown below.

And Or Xor

1x1=1	1+1=1	1xor1 = 0
1x0=0	1+0=1	1xor0 = 1
0x1=0	0+1=1	0xor1 = 1
0x0=0	0+0=0	0xor0 = 0

Let's find the logical result of the AX register in the program listed.

```
MOV AX,3545
AND AX,1722
```

1. Convert both numbers into binary and arrange them one over the other. (write them down in four bit groups)
2. Perform the logical function on each pair of bits, starting with the LSB.

```
  B15-|              B0-|
3545h=0011010101000101 ←(LSB)
1722h=0001011100100010
      0001010100000000 - (result)
```

3. Convert the result back into hex.

```
0001010100000000 = 1500h
```

Then AX = 1500.

```
NOT AX
NOT AL
NOT BYTE [120]
```

The not instruction will invert each bit of the specified register or memory location.(This will produce the one's complement)

```
AND AX,[120]
AND AX,F000
```

```
AND AX,BX
```

Produces the logical result of and-ing the two values specified. This is useful for resetting or clearing an individual bit in the program. (1x0=0)

```
OR AX,[350]
OR AX,1556
OR AX,BX
```

Produces the logical result of or-ing the two values specified. This is useful for setting an individual bit in the program. (1+0=1)

```
XOR AX,[500]
XOR AX,BX
XOR AX,AX
```

Produces the logical result of xor-ing the two values specified. This is useful for comparing two values.

```
SHL AX,CL
SHL AL,01
SHL AX,01
```

Shifts the bits of the value specified by the number of times in the CL register. SHL will shift to the left. SHR will shift to the right. The bits that are pushed out of the register are dropped and zero's are pushed into the other side.

```
ROL AX,CL
ROL AL,01
ROR AX,CL
ROR AL,01
```

Rotates the bits of the value specified by the number of times in the CL register. ROR rotates to the right. ROL rotates to the left. In the rotate, all the bits that are pushed out of the register, are rotated around to the other side and pushed back in. Suppose AX=834F and we have an instruction like,

```
ROR AX,1
```

The rotate is toward the right.

`AX = 1000 0011 0100 1111` ← the LSB rotates around to the other side

`MSB-->1 1000 0011 0100 111` (the bit pushed out will become the MSB)

If we regroup and convert to hex,

1100 0001 1010 0111b = C1A7h

AX = C1A7

How to use Transfer of control instructions

```
CALL 240
CALL [200]
CALL [BX]
```

Calls a sub-routine in another part of the program. The sub-routine must end with RET. When the sub-routine is finished, the program Will return to the instruction immediately following the call instruction.

```
RET
```

Must be the last instruction of a sub-routine. Causes the program to return to the calling program.

```
JMP 100

JMP [120]

JMP [BX]
```

This is an un-conditional jump to the specified address. The instruction at the specified location is executed and the program continues from that point.

```
JZ 150
```

This is a conditional jump to the specified address if the zero flag is set. (JZ=jump if zero). The address must be within 80 hex bytes of the JZ instruction. (short jump) This instruction must immediately follow an arithmetic, logical or INT instruction.

```
JNZ 300
```

This is a conditional jump to the specified address if the zero flag is not set. (JNZ= jump if not zero) The address must be within 80 hex bytes of the JNZ instruction. (short Jump) This instruction must immediately follow an arithmetic, logical or INT instruction.

```
LOOP 250
```

The loop instruction is used to execute an instruction or a group of instructions more than one time. Like the FOR/NEXT instruction in basic programming. The loop count is put into the CX register before the loop instruction. Each time the loop instruction is executed, the CX register will count down by one. When CX counts down to zero the program will go the next instruction after LOOP.

Example:

```
MOV CX,07D0     The count for the loop
MOV AH,9
MOV BH,50       This group of instructions
MOV BL,DL       will be repeated 7D0 times.
MOV AL,DL
INT 10
INC DL
LOOP 103        The program will jump back to offset 103 until CX=0
INT 20          After 7D0 times the program will end
```

Predict what this program does and than use the A command to assemble it. After the program is assembled, use the G command to run and verify the results.

```
INT XX
```

The INT instruction is used to access the operating system (IBMBIO and IBMDOS) as well as the ROM bios chip on the mother board. XX in this instruction is one of 256 possible INT calls.

What is an interrupt

There are two kinds of interrupts in the IBM personal computer, the hardware interrupt that is produced by an external device, such as a printer or a disk drive and the software interrupt that is produced by the INT instruction of a program.

Hardware interrupt

If you look at the pinout of the 8088 microprocessor, you will notice that it has a pin marked INTR (pin 18) that is called the interrupt request. When this pin becomes active, the microprocessor is interrupted from what ever it was doing so that it can execute a small interrupt service program to manage the type interrupt request that occurred. One pin for a hardware interrupt is not enough because of the many devices connected to the system bus through the I/O slots and support chips on the mother board.

8259 Interrupt controller chip

The 8259 interrupt controller chip is a very complex programmable IC circuit that manages up to 8 different interrupt inputs on the XT and with two cascaded 8259's up to 15 interrupt inputs on the AT type system board. The 8259 is basically an 8 input priority encoder that has 8 inputs and one output. The output is connected to the INTR pin on the micro-processor and up to 8 devices can be connected to the inputs. If more than one input became active at the exact same time, the input with the lowest number will be processed first, followed by the next and so on. It expands the number of INTERRUPT signals from one to eight with interrupt numbers 0-7.

The following chart shows the priority level of each input and what it is used for.

8259 input	Interrupt	Used for
IRQ0	INT 8	Time of day tick count
IRQ1	INT 9	Keyboard
IRQ2	INT	A Color graphic's adapter
IRQ3	INT B	Secondary serial adapter
IRQ4	INT C	Primary serial adapter
IRQ5	INT D	Hard drive (XT)
IRQ6	INT E	Floppy drive
IRQ7	INT F	Printer

Let's take a look at what happens when you press a key on the keyboard. We'll assume that you are at the DOS prompt and you want to enter the command DIR to do a directory of the active disk drive. While you are looking at the DOS prompt the microprocessor is executing many instruction inside the command.com program. One of the things that must happen is the updating of the time of day. As you know the computer keeps track of the time of day. It does this by counting a number of pulses that the clock generator circuit produces. The time of day is very important, in fact so important it is connected to the IRQ0 pin of the 8259 which has the highest

priority of the interrupt controller chip. Every 18 usec the clock circuit will produce an active signal on the IRQ0 line. When this happens the output of the 8259 goes active causing the INTR pin of the micro-processor to go active, interrupting the CPU. The CPU will finish the current instructions it was executing at the time and when finished will begin the process of saving the contents of all of its registers. To save the contents of the registers, the CPU PUSH'S all of them onto the stack, which is located in the high memory area of RAM. As soon as the CPU is finished preserving it's contents, it will signal the 8259 interrupt controller chip that it is ready to service the request by putting an active signal on the INTA (interrupt acknowledge) line of the control bus. As soon as the INTA signal is produced by the CPU, the 8259 interrupt controller will place a one byte address on the data bus that is used by the CPU to look up the address of a sub-routine program located inside the ROM bios chip that contains all the instructions to update the time of day bytes located in the lower RAM area of memory. After the time of day is up-dated, the CPU must POP the data for the registers back off of the stack. As soon as that is finished and the instruction pointer is pointing back to the program that was running at the time, the program starts to execute its instructions just as though nothing ever happened.

Block diagram of the 8088 micro-processor circuit. Note the 8259 interrupt controller chip and its 8 inputs.

Interrupt vector table

The interrupt vector table is a 1000 byte table of addresses, located at segment:offset 0000:0000 thru 0000:03FF, which is the lowest area of ram. Each entry in this table is a double-word or four bytes long that contains the segment:offset address of the sub-routine for any of 256 possible interrupt service numbers. Each interrupt address is 4 bytes long and it is stored in reverse order. You can calculate the offset into this table for any interrupt number by multiplying the interrupt number times 4. When an interrupt occurs the interrupting device must place a one byte offset on the data bus that will be used to jump to the proper offset for the real address of the interrupt service sub-routine. So, when an interrupt 0 occurs, the number 0 is put on the data bus, and the CPU will jump to offset 0000 to get the address of the interrupt service routine. (To calculate this, use the equation, Interrupt number X 4. For INT 0 it is 0 X 4 = 0) The following is a print out of the interrupt vector table on my 286 machine. To find the address for INT 10, multiply the interrupt number times 4. (10 X 4 = 40)

The four bytes starting at offset is the double-word address of INT 10 in reverse order.

```
LSB :MSB
```

`0000:0040  2D 1B C4 03` These are the bytes at offset 40.

`03 C4 1B 2D` when you reverse the bytes you will have the segment `03C4:1B2D` offset of the address

```
0000:0000  8A 10 23 01 F4 06 70 00-16 00 92 06 F4 06 70 00
0000:0010  F4 06 70 00 54 FF 00 F0-A6 EA 00 F0 A6 EA 00 F0
0000:0020  07 01 BC 07 99 01 BC 07-57 00 92 06 6F 00 92 06
0000:0030  87 00 92 06 9F 00 92 06-B7 00 92 06 F4 06 70 00
0000:0040  2D 1B C4 03 4D F8 00 F0-41 F8 00 F0 43 04 BC 07
0000:0050  39 E7 00 F0 FC 01 66 02-2E E8 00 F0 D2 EF 00 F0
```

```
0000:0060  00 E0 00 F0 2F 00 07 07-6E FE 00 F0 9F 01 BD 02
0000:0070  53 FF 00 F0 A4 F0 00 F0-22 05 00 00 F0 4C 00 C0
0000:0080  94 10 23 01 9E 10 23 01-6B 05 D1 08 A1 05 D1 08
0000:0090  33 05 D1 08 50 04 BC 07-5C 04 BC 07 BC 10 23 01
0000:00A0  6E 04 BC 07 03 05 BD 02-DA 10 23 01 DA 10 23 01
```

Software interrupts

A software interrupt is executed with an INT instruction in your program. It is similar to the CALL instruction in that, when it is executed the program will jump to another location in memory for the next instruction to be executed and when that group of instructions are finished a RET will return control to the calling program. The INT instructions are located inside IBMBIO and IBMDOS or in some cases inside the ROM bios chip. These instructions are basically sub-routines located inside the operating system.

Where are these sub-routines

The operating system is made up three different programs that are located in low memory starting at 00000 and ending at 0B000. The actual ending address will depend on the version of DOS and the number of device drivers that are loaded during the boot process. The block diagram above shows what the memory map of the IBM PC contains after the boot up.

The first 3FF bytes of RAM, starting at 00000, contain the interrupt vector table, which is a table of addresses that point to the INT xx sub-routines in IBMBIO, IBMDOS or the ROM BIOS chip. (see the discussion on the interrupt vector table)

The next FF bytes starting at 00400 are used by the operating system to store a complete equipment list in HEX form. If this Hex information is properly decoded, it can be used to determine what kind of equipment is connected to the system. Then starting at 00500 is the IBMBIO program that contains all of the sub-routines used by the operating system to interface with the hardware. BIO stands for BASIC INPUT/OUTPUT and it contains very low level instructions that communicate with such devices as the keyboard, printer, video, disk drives and the chips on the mother board.

The next program loaded in is called IBMDOS, which is a little more complex and it contains sub-routines that interface with the DISK OPERATING SYSTEM. The instructions in IBMDOS are communicating with the IBMBIO program and the last program loaded into memory, COMMAND.COM. Command.com is the command interpreter that monitors the keyboard and waits for you to press the enter key. When the enter key is pressed it is command.com's job to determine what command was entered. The command will either be an internal command or an external command, depending on where the instructions for the command are stored. All internal commands are located inside command.com itself and all external commands are located on the disk drive. After the command is interpreted, command.com will pass on all the parameters to the IBMDOS program. IBMDOS will process the parameters into the proper format for the IBMBIO program, which will actually turn on the drive or control the hardware needed for the command entered.

MS-DOS is a three level operating system made up of three programs that are at three levels of programming. Command.com is the highest level because it understands commands like DIR, DATE, COPY etc. IBMDOS is the second highest

level because it receives instructions from command.com and passes them on to IBMBIO which is the lowest level of programming. All of the sub-routines that are part of these three programs and written by Micro-soft and IBM, perform specific functions in the computer and its hardware. For example, there are sub-routines that were written just to control the video monitor and how it displays data on the screen. These routines are used to move the cursor, change the color of the characters that are displayed, change the character size, print strings of text and many other display functions. Other sub-routines control every piece of hardware attached to the system. The sub-routines used in the operating system require many 8088 instructions to perform a specific task and are very complex in there program style. Fortunately, IBM and Micro-Soft designed a method that allows the computer programmer to utilize all of these tested and proven sub-routines in their own programs. This makes programming a little easier for a programmer because the he/she will spend less time writing code that has already been written by the most knowledgeable programmers in the country. The 8088 was designed with a special instruction called INT that will execute any one of these sub-routines from within our program. INT is what we call a software interrupt, because when it is executed from a program, the micro-processor is interrupted for a time while the sub-routine in IBMBIO or IBMDOS is executed. When the sub-routine is finished the micro-processor can go back to the calling program and finish the remainder of its instructions .

How can we use the interrupts

To use a software interrupt, the programmer must set-up the program before the INT instruction is actually executed. Most interrupts require certain values for the proper operation of the interrupt sub-routine. In other words, all interrupts require an input from your program and it must be stored inside specific registers. The instruction that will actually execute the sub-routine is the INT xx instruction, where xx is a number from 0 - FF, depending on the interrupt you want to execute.

In order to use these interrupts you must make reference to the interrupt list that is supplied by IBM and Micro-Soft. It is not practical to rewrite this list in this book because of the size of its contents. I would recommend that you purchase a book that contains some of the more common interrupts. This book will, however, list some of the most used interrupts in programming.

The interrupts used in programs are going to access either IBMDOS or IBMBIO and you should know the difference before you use them. In general, any interrupt that calls IBMDOS is going to execute a little slower because, as was stated before, IBMDOS is a higher level program and its instructions are passed onto IBMBIO which actually turns on the chips. Going thru IBMDOS first, slows down the program but because it is a higher level, the input required from your program is much easier to develop and set-up. DOS interrupts are often referred to as DOS CALLS and they are all INT 2x. The other interrupt type is often called a BIOS interrupt because it calls sub-routines inside IBMBIO or the ROM BIOS chip. These interrupts are much faster and can speed up a program considerably but, because the BIO is a low level program, the programmer must understand the hardware in the system and supply more input from his program.

The following chart shows the most used interrupts.

Bios Calls | Dos Calls

10 | 14 20 | 24

11 | 15 21 | 25

12 | 16 22 | 26

13 | 17 23 | 27

14 | 19 1A |

All of these Interrupts are sub-routines in one of the operating system programs and they are further divided into smaller sub-routines called Functions and Sub-Function. For example, INT 10 has a total of 55 Functions and sub-Functions within its code. To access a Function within an Interrupt, The calling program must have the Function number inside the AH register.

The AH register is used exclusively for the Function.

Let's write a program that uses an INT instruction, to show how in must be set-up. Our program will put the cursor in the middle of the screen and print the letters CPI. A look through the interrupt list reveals two interrupts that will be needed for this program. INT 10 Function 2 and INT 10 Function 9.

INT 10 Function 2 - Set the cursor position

Input Returns

AH = 02 cursor is moved

BH = page number

DH = row (y coordinate)

DL = column (x coordinate)

The x-y coordinate of an 80x25 CGA display is shown below.

INT 10 Function 9 - Write character and attribute at the cursor position

Input Returns

AH = 09 Character displayed

AL = Hex ASCII code for character

BH = page number

BL = attribute byte

CX = number of characters to write

Attribute byte

```
 B7 B6 B5 B4 B3 B2 B1 B0
|x |R |G |B |x |R |G |B |
------------ -----------
       back | fore
     ground | ground
      color | color
```

The attribute byte is used to set the color of the background and foreground of the characters that are to be displayed. R,G,B are the red, green and blue guns on the Cathode ray tube (CRT). A set bit will turn on the specified gun and produce the desired color. There is a separate RGB bit's for the foreground and background colors.

BITS 0-1-2 control the foreground color.

BITS 4-5-6 control the background color.

BIT 3 controls intensity (1=high)

BIT 7 controls blinking (1=blink)

To produce a blue background and a white foreground

bits 4-2-1-0 will have to be set to a 1.

all other bits are cleared to 0.

The binary number than would be,

0001 0111 binary

or

17h Hex

To do this in a program than BL would have the Hex number 17. We will begin this program by witting down the necessary steps in the order of execution.

1. Position the cursor to the middle of the screen.
2. Display the letter C.
3. Position the cursor for the next character
4. Display the letter P.
5. Position the cursor for the next character
6. Display the letter I.

This is a simple straight line type program because it is simply going to execute instructions in sequential order starting at offset 0100 in memory. The steps will position the cursor, display the letter. Position the cursor, display the letter etc.

```
0100 Mov AH,2--------|
0103 Mov DH,C------- | Step 1
0105 Mov DL,24------ | Position the cursor
0107 Mov BH,0------- | row C, column 24
0109 INT 10
0000 Mov AH,9------- |
0000 Mov AL,43------ |
0000 Mov BH,0------- | Step 2
0000 Mov BL,17------ | Display the letter C
0000 Mov CX,1--------| Blue background White foreground
0000 Int 10----------|
0000 Mov AH,2--------|
0000 Mov DH,C--------| Step 3
0000 Mov DL,25-------| Position the cursor to
0000 Mov BH,0--------| row 25, column 25
0000 Int 10 ---------|
0000 Mov AH,9--------|
```

```
0000 Mov AL,50-------|
0000 Mov BH,0--------| Step 4
0000 Mov BL,17-------| Display the letter P
0000 Mov CX,1--------| Blue background
0000 Int 10----------| White foreground
0000 Mov AH,2--------|
0000 Mov DH,C--------| Step 5
0000 Mov DL,26-------| Position the cursor to
0000 Mov BH,0--------| row C, column 26
0000 Int 10 ---------|
0000 Mov AH,9--------|
0000 Mov AL,49-------|
0000 Mov BH,0--------| Step 6
0000 Mov BL,17-------| Display the letter I
0000 Mov CX,1--------| Blue background White foreground
0000 Int 10----------|
0000 Int 20----------|End the program
```

To position the cursor, the program executes an INT_10 Function 2. The values needed in the program are listed below,

AH = 2 For the function number

BH = 0 For the page number (0 is the 1st page out of 4) in video memory

DH = C The row number in Hex

DL = 24 The column number in Hex

INT 10 is the instruction that actually makes the call to IBMBIOS.

The same code is used in steps 3 and 5 but the column number in DL is advancing to the next character position.

To display a character, the program executes an INT_10 Function 9. The values needed in the program are listed below,

AH = 9 For the function number

BH = 0 For the page number

AL = 43 The Hex ASCII code for the letter C

BL = 17 The attribute byte that will cause a blue background and a white foreground

CX = 1 For the number of characters to display

The same code is used in steps 4 and 6 but the AL register is different for each letter to be displayed.

This program was written this way to demonstrate how to set-up the registers in your code before the INT instruction is executed. The order that the registers are set-up in the code is not important as long as it is done before the INT. A much more efficient program will be shown later to explain some of the advanced instructions of the 8088.

Boot process

The boot-up process begins when the reset circuit in the clock generator chip produces the reset pulse that is applied to pin 21 of the 8088 CPU. This pin is connected to the reset circuit inside the micro-processor to force a reset address on address bus A0 thru A20. This will execute a jump instruction at address F000:FFF0 inside the ROM bios chip that points to the first instruction of the BIOS.

The ROM bios program is approximately 8K bytes long, and controls all of the hardware on the system board and interface cards. The CPU support chips are initialized with the proper default values to control such things as the video monitor, disk drives, printer ports and keyboard. After the initialization of all the hardware, the program executes a very extensive diagnostic type test on the 8088, ROMS, RAM etc. to complete what is called the POST or Power On Self Test.

If there are no critical errors during the POST, the A drive is turned on and tested for an open or closed door. The closed door condition will cause the head to position over track 0, head 0, sector 0 of the disk in the A drive and the boot loader program is transferred into memory. Once in memory, the boot loader program is given control of the CPU and a series of instructions are executed that will look in the directory of the disk for the system files, IBMBIO and IBM DOS. If these two system files are on the disk, they are loaded into low memory in that order, along with any driver programs that are listed in the device= statement of config.sys. Control of the CPU is then given to the IBMDOS program to finish the boot up process, by loading COMMAND.COM into memory in the next available space right after IBMDOS.

The boot process is complete when COMMAND.COM is given the final control of the CPU and the A prompt appears. If the system files do not exist or they are corrupt in any way, the boot loader program will display the familiar message " DISK BOOT FAILURE ".

An open door in the A drive will cause the boot process to continue from the C drive if it exist.

Debugging a program

Debug is very useful to use when debugging a program or running the program for the first time.

When a program has an error in the code and it is executed, most likely the computer will lock up making it necessary for you to reboot the machine. This can be prevented by executing only a small part of the code at a time to verify that it is OK and it does what you expect it to do. Below is the code of a small program and an explanation of how I would use the address and breakpoint parameters to debug it.

Assemble the following program and save it to the disk with the name BREAK.COM if you intend to try any of the following. In order to use breakpoint for debugging you must reload the program from the disk each time the program terminated normally.

```
CS:0100 MOV AH,9 -----|
CS:0102 MOV DX,200 ---| print a message on the screen
CS:0105 INT 21 -------|
CS:0107 MOV AH,7 -----| read the keyboard and put ASCII
CS:010B INT 21 -------| code into the al register.
CS:010B CMP AL,1B ----| see if it was the escape key
CS:010D JZ 0112 ------| and if it was end the program
```

```
CS:010F JMP 108 ------| if it wasn't the escape, read another
                        key
CS:0111 INT 20 -------| end the program and return to DOS
```

This is a very simple program that will print a short message on the screen. The message is stored in memory at offset 200 and the last character of the message is the dollar sign ($) or HEX 24. The HEX 24 ($) is what we call the end of file marker (EOF).

After the message is displayed properly, the program continues to the next part of the program, which will pause and wait for us to press a key on the keyboard. When we press any key the ASCII code for that key is stored inside the al register where it will be used later.

When a key is pressed the program will test it to see if the ASCII code is equal to HEX 1B. If the Test proves to be true (equal) the program will terminate and return to DOS.

If the test proves not true (not equal) then the program jumps up to the instructions that will read a key. This program repeats until We press the ESCAPE key on the keyboard.

I have divided the program up into blocks of code that will perform a function when executed, the functions in this program are:

1. Print a message on the screen.(offset 100-107)
2. Read a key from the keyboard. (offset 107-10B)
3. Test the key for HEX 1B. (offset 10B)
4. End if it is 1B. (offset 10D)
5. Get another key. (offset 10F)
6. End (offset 111)

Hot to debug a program

To debug the program we are going to use the P command and the G command to either single step through the program one instruction or interrupt at a time or a group of instructions up to a specific offset. After breaking the code up into groups of instructions that perform a function, we can execute them to verify proper operation. Refer to the program above for the next few steps.

The instructions from 0100 to 0106 will display a message on the screen when the INT 21 instruction is executed. To test this group of instructions use the G command to set break-points used by debug to control the flow of the program. Reset the IP register to 0100 and then enter,

```
G=100 107 (return)
```

This will execute all the instructions starting at offset 0100 and ending at offset 0107. If the code in this part of the program is valid, the message that is stored at offset 200 will be displayed on the screen.

The G command is being used with a start address followed by an end address that stops the micro-processor from executing pass a specific instruction. (This is called a break-point)

If for some reason the message was not displayed on the screen, we could conclude that there is a problem with the code in that group of instructions. To test the

instructions one at a time, we will reset the IP register back to 0100 and then enter the P command.

```
P (return)
```

The P command is very similar to the T (trace) command, in that only one instruction at a time will be executed. However, the P command will execute an INT instruction, Where as the T command will not. Either one could be used here since the instruction being tested is a simple move instruction which will be verified by checking the AH register after the MOV AH,9 instruction is executed. The P and T command show the contents of the registers after the instruction is executed. The IP register is updated by the micro-processor and will be pointing to the next instruction in the program. If this instruction worked properly and is valid enter,

```
P (return)
```

This will execute the next MOV instruction at offset 102. Again, we will check the registers to verify the MOV DX,200 instruction. Enter,

```
P (return)
```

and the INT instruction at offset 0105 is executed to display the message. This process should continue until you locate the code that is not working properly in the program. Enter,

```
G=100,107 To test the message code.
G=107,10B To test the keyboard input.
G=10B,10F To test the compare instruction.
G=10F To test the jump instruction.
```

Power on self test

Number of test	Description working Failure
1	8088 processor test none halt at FEOAD
2	BIOS ROM check sum none halt at FE0AD
3	Timer-1 test speaker click halt at FE0AD
4	Initialize Timer-1 none
5	Test DMAC channel-0 none halt at FE12D
6	Initialize DMAC 0 none halt at FE15C for refresh
7	Test 1st 16K of Ram none system halts
8	Initialize PIC none
9	read configuration none
10	Init CRT and test cursor beep 1long 2short video Ram appears
11	Test PIC none "101" halt at FE35C

12	Test PIT channel-0 none "101" halt at FE35C
13	Keyboard test none "301"+scan code
14	Set INT table none
15	Test expansion
16	Test all Ram display DMA failure
17	Test refresh xxxK OK "101" halt at FE35C
18	Check for ROM basic none Display "ROM"
19	Test disk drv. adpt. motor on Display "601"
20	Init ser/par printer none
21	Display error message cls "error resume=F1" 2 short beeps
22	Enable NMI none "Parity check 1" "Parity check 2" halt at FE8EF
23	Boot DOS from disk DOS prompt Boot from ROM basic

Creating a COM program that will display a text file on the screen.

Step 1. Using any word processor, generate the text file that you would like to be displayed and end it with the dollar sign character ($).

Return to the top of the text file and enter one line of spaces that will reserve 80 bytes of the file to be used later. Save it to the disk as text.txt.

Step 2. Load the text.txt file into debug, using the command

DEBUG TEXT.TXT. Using the dump command, verify that the file is entered properly in memory. You should notice that the first 50 hex bytes are 20, the ASCII code for the space character.

When the file was created you entered 80 spaces, which is equal to 50 hex spaces.

Step 3. Assemble the following code starting at offset 100.

```
XXXX:0100 MOV AH,09
MOV DX,150
INT 21
INT 20
```

Step 4.Using the name command, enter the new name with a .COM extension. (TEXT.COM). Find the offset number for the end of the file ($) and set the CX register for the number of bytes.

Enter w to write the file to the disk. Quit debug and enter the name of the com file you just assembled. If the program is working ok, it will be displayed on the screen.

How to assemble a COM program.

After you have converted the flow chart of your program into the instruction code on paper, you are ready to begin the assembly process using Debug as the assembler. At the debug prompt (-) enter A 100 (return). You should see the segment-offset on the left side of the screen. We start assembly of all COM programs at offset 100 because the first 100 hex bytes is reserved for the Program Segment Prefix.(0-FF) If there isn't a valid instruction at offset 100, the program will cause the computer to crash, requiring a reboot of the system.

Follow these steps to assemble a program that will display the ASCII code on the screen in a different way.

```
A100 (return)
XXXX:0100  MOV CX,7D0 (return)
XXXX:0103  MOV AH,09 (return)
XXXX:0105  MOV BH,50 (return)
XXXX:0107  MOV BL,DL (return)
XXXX:0109  MOV AL,DL (return)
XXXX:010B  INT 10 (return)
XXXX:010D  INC DL (return)
XXXX:010F  LOOP 0103 (return)
XXXX:0111  INT 20 (return)
XXXX:0113  (return)
```

When in the A or assemble mode, debug will automatically calculate the offset for the next instruction. After the first instruction is entered at offset 100, debug displays the next offset available for the next instruction. In this program the instruction MOV CX,7D0 requires 3 bytes of memory starting at 0100 and going to 0102. The next instruction (MOV AH,09) will start at offset 0103.

The last instruction of the program must be an INT 20 in order to end the program properly. This is a two byte instruction located at offset 0111. This is determined by observing the offset of the next available instruction which is at 0113.

To write the program to the disk, you must name it with the N command, set the file size with the CX register and then enter W to do the actual writing to the disk. Enter,

```
_n a:\ascii.com (return)
_R cx
:13 (return)
_w (return)
```

We entered the complete path for the file name so that it would be written to a specific disk and directory. (A:\)

The file size was determined by subtracting 100 from 113, which is the next byte after the last instruction of the program. We subtract from 100 because that is the offset that the program started at. (113 - 100 = 13)

Since the name and file size is entered properly, the last step is to simply enter W.

If all the steps were done properly, you can quit debug (q) and at the DOS prompt enter DIR to see if the file is listed in the directory. Notice that the file size is 19 byte decimal or 13 bytes hex.

Enter ASCII (return) and the program should execute and display the ASCII code from the bottom of screen to the top.

Getting the keyboard status byte

Interrupt 16 Function 2 will return one byte of information about the status of certain keys on the keyboard. The keys that are tested for their status are the Insert, Caps lock, Num lock, Scroll lock, Alt, Ctrl, Left shift and the right shift. The interrupt indicates whether or not the keys are active by setting a bit in the AL register on return. If a bit is set (logic one), than the key is pressed, a clear bit (logic 0) indicates that the key is not pressed.