debug: Learn how to crack windows, programs ect manually

Debug is a program that comes with modern versions of DOS (I do not know when I started shipping out with DOS). Anyway, all Windows users should have it already.

It's a great tool for debuging programs, unassembling and cracking, and reading "hidden" memory areas like the boot sector, and much more.

The following was copied from an assembly tutorial who's author we cannot credit, because we have no idea who he is.

Get into DOS and type "debug", you will get a prompt like this:

```
-
```

now type "?", you should get the following response:

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

Lets go through each of these commands:
Assemble:

```
-a
107A:0100
```

At this point you can start assembling some programs, just like using a assembler. However the debug assembler is very limited as you will probably notice. Lets try to enter a simple program:

```
-a
107A:0100 MOV AH,02
107A:0102 MOV DL,41
107A:0104 INT 21
107A:0106 INT 20
-g
A

Program terminated normally
```

That's the same program we did at the end of the previous chapter. Notice how you run the program you just entered with "g", and also notice how the set-up part is not there? That's because debug is just too limited to support that.
Another thing you can do with assemble is specify the address at which you want to start, by default this is 0100 since that's where all .COM files start.
Compare:

Compare takes 2 block of memory and displays them side by side, byte for byte. Lets do an example. Quite out of debug if you haven't already using "q". Now type "debug c:\command.com"

```
-c 0100 l 8 0200
10A3:0100 7A 06 10A3:0200
```

This command compared offset 0100 with 0200 for a length of 8 bytes. Debug responded with the location that was DIFFERENT. If 2 locations were the same, debug would just omit them, if all are the same debug would simply return to the prompt without any response.
Dump:

Dump will dump a specified memory segment. To test it, code that assembly program again:

```
C:\>debug
-a
107A:0100 MOV AH,02
107A:0102 MOV DL,41
107A:0104 INT 21
107A:0106 INT 20
-d 0100 l 8
107A:0100 B4 02 B2 41 CD 21 CD 20
...A.!.
```

The "B4 02 B2 41 CD 21 CD 20" is the program you just made in machine language.

```
B4 02 = MOV AH,02
B2 41 = MOV DL,41
CD 21 = INT 21
CD 20 = INT 20
```

The "...A.!." part is your program in ASCII. The "." represent non-printable characters. Notice the A in there.
Enter:

This is one of the hard commands. With it you can enter/change certain memory areas. Lets change our program so that it prints a B instead of an A.

```
-e 0103 <-- edit program at segment 0103
107A:0103 41.42 <-- change 41 to 42
-g
B

Program terminated normally
-
```

Wasn't that amazing?

Fill:

This command is fairly useless, but who knows....

It fills the specified amount of memory with the specified data. Lets for example clear out all memory from segment 0100 to 0108, which happens to be our program.

```
-f 0100 l 8 0 <-- file offset 0100 for a length of 8 bytes with 0
-d 0100 l 8 <-- verify that it worked
107A:0100 00 00 00 00 00 00 00 00 .......
```

Yep, it worked.

Go:

So far we used go (g) to start the program we just created. But Go can be used for much more. For example, lets say we want to execute a program at 107B:0100:

```
-r CS <-- set the CS register to point to 107B
CS 107A
:107B
-g =100
```

You can also set breakpoints.

```
-a <-- enter our original program so we have something
107A:0100 MOV AH,02 to work with
107A:0102 MOV DL,41
107A:0104 INT 21
107A:0106 INT 20
-g 102 <-- set up a break point at 107A:0102
```

At this point the program will stop, display all registers and the current instruction.

Hex:

This can be very useful. It subtracts and adds two hexadecimal values:

```
-h 2 1
0003 0001 <-- 2h + 1+ = 3h and 2h - 1h = 1h
```

This is very useful for calculating a programs length, as you will see later.

Input:

This is one of the more advanced commands, and I decided not to talk about it too much for now. It will read a byte of data from any of your computers I/O ports (keyboard, mouse, printer, etc).

```
-i 3FD
60
-
```

Your data may be different.

In case you want to know, 3FD is Com port 1, also known as First Asynchronous Adapter.

Load:

This command has 2 formats. It can be used to load the filename specified with the name command (n), or it can load a specific sector.

```
-n c:\command.com
-l
```

This will load command.com into debug. When a valid program is loaded all registers will be set up and ready to execute the program.
The other method is a bit more complicated, but potential also more usefull. The syntax is

```
L <address> <drive letter/> <sector> <amount to load>
-l 100 2 10 20
```

This will load starting at offset 0100 from drive C (0 = A, 1 = B, 2 = C, etc), sector 10h for 20h sectors.
This can be useful for recovering files you deleted.
Move:

Move takes a byte from the starting address and moves it to the destination address. This is very good to temporary move data into a free area, than manipulate it without having to worry about affecting the original program. It is especially useful if used in conjunction with the r command to which I will get later. Lets try an example:

```
-a <-- enter our original program so we have something
107A:0100 MOV AH,02 to work with
107A:0102 MOV DL,41
107A:0104 INT 21
107A:0106 INT 20
-m 107A:0100 L 8 107B:0100 <-- more 8 bytes starting from 107A:0100 into 107B:0100
-e 107B:0103 <-- edit 107B:0103
107B:0103 41.42 <-- and change it 42 (
-d 107A:0100 L 8 <-- make sure it worked
107A:0100 B4 02 B2 41 CD 21 CD 20 ...A.!.
-d 107B:0100 L 8
107A:0100 B4 02 B2 42 CD 21 CD 20 ...B.!.
-m 107B:0100 L 8 107A:0100 <-- restore the original program since we like the changes.
```

Name:

This will set debug up with a filename to use for I/O commands. You have to include the file extension, and you may use addition commands:

```
-n c:\command.com
```

Output:

Exactly what you think it is. Output sends stuff to an I/O port. If you have an external modem with those cool lights on it, you can test this out. Find out what port your modem is on and use the corresponding hex number below:

Com 1 = 3F8 - 3FF (3DF for mine)
Com 2 = 2F8 - 2FF
Com 3 = ??? - ??? (if someone knows, please let me know)

Now turn on the DTA (Data Terminal Ready) bit by sending 01h to it:

```
-o XXX 1 <-- XXX is the com port in hex
```

As soon as you hit enter, take a look at your modem, you should see a light light up. You can have even more fun with the output command. Say someone put one of those BIOS passwords on "your" computer. Usually you'd have to take out the battery to get rid of it, but not anymore:

```
MI/AWARD BIOS
-o 70 17
-o 71 17

QPHOENIX BIOS
-o 70 FF
-o 71 17

QGENERIC
-o 70 2E
-o 71 FF
```

These commands will clear the BIOS memory, thus disabling the password.
Proceed:

Proceeds in the execution of a program, usually used together withy Trace, which I will cover later. Like the go command, you can specify an address from which to start

```
using =address
-p 2
```

Debug will respond with the registers and the current command to be executed.
Quite:

This has got to be the most advanced feature of debug, it exits debug!

```
-q
```

Register:

This command can be used to display the current value of all registers, or to manually set them. This is very useful for writing files as you will see later on.

```
-r AX
AX: 011B
:5
-
```

Search:

Another very useful command. It is used to find the occurrence of a specific byte, or series of bytes in a segment. The data to search for can by either characters, or a hex value. Hex values are entered with a space or comma in between them, and characters are enclosed with quotes (single or double). You can also search for hex and characters with the same string:

```
-n c:\command.com <-- load command.com so we have some data to search in
-l
-s 0 l 0 "MS-DOS" <-- search entire memory block for "MS-DOS"
10A3:39E9 <-- found the string in 10A3:39E9
```

NOTE: the search is case sensitive!
Trace:

This is a truly great feature of debug. It will trace through a program one instruction at a time, displaying
the instruction and registers after each. Like the go command you can specify where to start executing from,
and for how long.

```
-a <-- yes, this thing again
107A:0100 MOV AH,02
107A:0102 MOV DL,41
107A:0104 INT 21
107A:0106 INT 20
-t =0100 8
```

If you leave out the amount of instructions that you want to trace, you can use the proceed (p) to continue
the execution as long as you want.

Unassemble:

Unassembles a block of code. Great for debugging (and cracking)

```
-u 100 L 8 <-- unassembles 8 bytes starting at offset 100
107A:0100 MOV AH,02 <-- debut's response
107A:0102 MOV DL,41
107A:0104 INT 21
107A:0106 INT 20
```

Write:

This command works very similar to Load. It also has 2 ways it can operate: using name, and by specifying an
exact location. Refer to back to Load for more information.

NOTE: The register CX must be set the file size in order to write!
NOTE: Write will not write .EXE or .HEX files.[SIZE=7][SIZE=14]