(hex and such by proZaq continued)

--==< The Computer's Hardware Components >==--

Now we have covered a lot of track. You should know what the different number systems are,
you should have an understanding of different programing expressions and you should know how to use a HexEditor. In order to understand how a Debugger works, however, we need to
dive into the hardware components of a computer. Do not worry, I will keep it simple. I will only talk about the most important parts of the computer. As a matter of fact you will most likely recognize and already know the function of some of these components.

- The Motherboard - This is that green board within your computer covered with circuits where all the hardware is placed. Everything from the diskdrive to the microphone is somehow connected to the Motherboard.
- Memory - The part of the computer where data is stored.
- RAM (Random Access Memory) - This is the temporary storage facility of the computer. It is loaded full with information when you turn on the computer and it is emptied when you turn your computer off.
- PRAM (Parameter RAM) - Very much like ordinary RAM with the exception that there is a special battery in the computer providing the PRAM with enough electricity to keep the information in it even when the rest of the computer is turned off.
- ROM (Read Only Memory) - This is a storage unit where information can only be read from. With other words the computer can read anything in the ROM but it can not change anything there. Thus the ROM usually stores all the information the computer needs to be able to start when you press the "On" button. When you think about it, a compact disk (CD) is also a read-only unit. The computer can read the information on it, but it can't store stuff on it. Thus the name CD-ROM.
- Storage Media - For example, the HardDrive, a floppy disk, a Zip disk, etc. These are accessories to the computer on which information is stored "indefinitely". That is "indefinitely" in the sense that the information will still be there, even when the computer off. This does, however, not keep the computer from replacing the information on the media. So it can freely read from it and write to it. It can even replace existing data with new data.
- The Processor - This is the brain of the computer. All data is sent to be calculated in the processor.
- Registers - These are blocks of memory within the processor where data is stored for a brief period of time, waiting to be processed.
- Busses - Circuits, on the motherboard, where the information travels from one component of the motherboard to the other.
- The Sound Card - This is like a small mother board with it's own processor, busses and registers capable of converting binary information to sound waves.
- The Graphics Card - Same as the sound card except it displays information as the graphics on the monitor.

And that's all you need to know for now.

--==< Debuggers >==--

I gave this whole chapter a lot of thought and decided on the following. I will only give a general description of a debugger, and some theoretical uses for it. There are a lot of different debuggers out there, for all computer platforms. I use a Macalong with Apple's own free debugger called MacsBug. For those interested I have included a file called "MacsBug". I wrote this file a while back and it is not designed to be read by beginners. However, some people might find it handy. There are a quite a lot of other files dealing with MacsBug that might be a lot more useful. So if you are really interested, read a few of those as well!

A debugger is a program that allows you to take control of the complete computer. This is done, by "stoping" the processor. When you activate a debugger, it stops the processor from executing any commands of the program that is running at the moment. The whole concept of a debugger is to help software developers look for mistakes in their programs or to see if it executes in a proper way.

As you may know, when a program is launched, the Operating System loads the program from
the HardDrive to the RAM. This is done because the RAM is a lot faster than the HardDrive and most other storage medias. Then the processor jumps to the part of the RAM where the code of the program is stored, and it starts executing each of the commands. So, when the debugger is started the processor stops executing these commands. It then allows the user to check the values of the certain hardware components. This way the user can detect any mistakes in the program or just check the current state of the hardware components. The user can even step through the code of the program. This means that they can look at each command that makes up the program and see what it does. As I said before, debuggers are largely used by programmers trying to figure out why their program won't work properly.

For you, the main advantage of a debugger will be that it allows you to change the data in most hardware components, including the RAM. Since the program is loaded into the RAM when
launched, and it does all the calculations in the RAM all the data/variables/information it may use will most likely be stored somewhere in the RAM. Thus the debugger can be used to
change any of these.

Since this file has had a general undertone of being an aid for cheating on computer games I decided to include a way to use MacsBug to cheat on games while you are actually playing them. I will first summarize the theory and then go into the specifics. In order for you to be able to follow it through you will have to know at least the basic commands and functions of MacsBug. If you are using a different debugger, then the theory will most likely be the same but the commands will be different.

WARNING: When you are changing memory contents or changing anything in a debugger for
that matter, you CAN cause very large damages to your computer! The incorrect use of a debugger can cause the computer to freeze and cause information to be lost! Several other damages
can also occur. Thus I advise you to become familiar with your debugger before you attempt to change anything with it. Read any related files, read the manuals and do some minor experimenting before you try to change stuff directly in the memory!

So first, the theory. I launch the game as a start. By opening up any saved games, I force the game to load anything it might have saved on the HardDrive (and that is of use to me) to the RAM. Then I stop the game by starting the debugger, I find where in the RAM the game is stored, I find the information I want to change and then I change it.

OK, and now for practice:

Here's the scenario: I'm playing Heroes of Might & Magic II and I want more creatures in my armies. I open up the hero's preference window and see that my hero has 25 Minotaurs, 53 Dwarfs, 32 Griffins, 9 Skeletons and 2 Dragons. Thus I know that the computer keeps track of how many creatures I have and that means that the number of creatures must be stored somewhere in the RAM.

The first thing I have to do is to find out where in the RAM the game is located. The first step is to drop into MB (this is done by holding down the apple key and pressing the power button on the keyboard).

The second step is to issue the "hz" (heap zone) command that lists all the currently active applications and their locations in the RAM. I got this:

```
 Heap zones
   #1   Mod          7206K   00002800 to 0070C34F   SysZone^
   #2   Mod             6K     00008D60 to 0000A88F   ROM read-only zone
   #3   Mod            48K     001301F0 to 0013C1EF
   #4   Mod           128K     004475B0 to 004675AF
   #5   Mod         29560K   0070C350 to 023EA69F   Process Manager zone
   #6   Mod          9737K     010A6830 to 01A28EFF   “Heroes II”       ApplZone^
TheZone^   Targ
```

As you can see Heroes II starts at memory location 010A6830 and ends at 01A28EFF (all in hex of course).

The next step is to use the "find" command and find the number of creatures that make up my army. See, it is very likely that a game stores relevant data close to each other. So I presume that the program stores the number of creatures I have, in a specific block of memory in the RAM. If I can find this block of memory, I will be able to change it's content, thus changing the number of creatures. In some ways it's like finding information with a HexEditor. Except you're looking for data in the RAM and not in a file.

In order to be able to use the "find" command I have to be able tell the following things: the start of the memory address, how many bytes the debuggers should search for, and what
to search for. Unfortunately I don't have all the criteria. I have to find out how many bytes Heroes II occupies. This can easily be done by subtracting $010A6830 from $01A28EFF. This subtraction gives me $009826CF. If you want you can do this calculation directly in MB, just type "01A28EFF-009826CF". Now I have all the stuff I need to use the find command.

In this example I issue "f 010A6830 009826CF 00190035"

The "f" stands for "find". This tells MB to use the find command.
"010A6830" is the address of the memory where Heroes II starts.
"009826CF" stands for the number bytes Heroes II occupies in the memory. It tells MB the number of bytes I want to search for, from the initial address.
"00190035" stands for 25 Minotaurs and 53 Dwarfs. #25=$19 and #53=$35. Since I've done
this before I know that Heroes II stores the the number of creatures in word sized blocks of memory. If I didn't know that I would have had to search for "0019" first (or "19") and look at it in it's context. When I issued the find command I got this:

```
Searching for 00190035 from 010A6830 to 01A28EFE
   0118EE3E   0019 0035 0020 0009   0002 0000 0003 0100   •••5•••••••••••
```

The first hex long represents an address in the memory.   The following four longs (16 bytes) are the values of the data contained in the RAM starting from that address.   The following 16 characters are the ASCII representations of these values.

Now, if I convert the first five words to decimal numbers I get: 25, 53, 32, 9 and 2. That's a perfect match of the number of creature I have in my army.   Thus there is a fairly good chance that HeroesII keeps track of my army starting at address 0118EE3E. When you are doing something like this on your own and you don't think that this is the location of the memory that you are looking for, you can continue searching by hitting return until you get a message saying that it could not be found.

Then comes the dangerous part, I have to change the value in the memory.   I issue the following command, "sw 0118EE3E 00ff" (sw stands for set word).   This changed the word at the memory address 0118EE3E from "0019" to "00ff".   If I now issue the "dm 0118EE3E" command (dm stands for display memory) I see that the value at address 0118EE3E has changed to:

```
   0118EE3E   00FF 0035 0020 0009   0002 0000 0003 0100   •••5•••••••••••
```

So I return to the game by issuing the "g" command.   Apparently nothing has changed.   But if I close the preferences window and force the game to actually check how many Minotaurs my army has (by checking the variable stored in the RAM), then I can see that the game in fact thinks that I have 255 Minotaurs!   Cheat accomplished.   Now I just have to repeat the above procedures for all the other creatures.

NOTE: when you are changing the contents of the memory, make sure that you use the appropriate addresses, meaning the ones you get when you issue "hz" and the find command.
Do NOT use the memory addresses I used!   They are purely examples and WILL NOT work on
your computer!

--==< End Notes >==--

In conclusion I hope to have given an insight to how different principles of computer technology work.   I'd like to point out to some more advanced readers that I am aware of the fact that I have generalized and simplified some concepts.   I did this only when I felt that the theory was more important than a detailed explanation.   I also hope to have made some of you interested in learning about programing and more advanced topics of IT and computer technology.   If you have any questions or comments you can reach me at prozaq@usa.net.

Good luck

ProZaq
1999.12.31

Werd to mSEC, and everyone else who's ever helped me out!   It's people like you who make it worthwhile!

The most common ASCII characters (to be viewd in Monaco)

32 ' ' $20	33 '!' $21	34 '"' $22	35 '#' $23	36 '$' $24
37 '%' $25	38 '&' $26	39 ''' $27	40 '(' $28	41 ')' $29
42 '*' $2A	43 '+' $2B	44 ',' $2C	45 '-' $2D	46 '.' $2E
47 '/' $2F	48 '0' $30	49 '1' $31	50 '2' $32	51 '3' $33
52 '4' $34	53 '5' $35	54 '6' $36	55 '7' $37	56 '8' $38
57 '9' $39	58 ':' $3A	59 ';' $3B	60 '<' $3C	61 '=' $3D
62 '>' $3E	63 '?' $3F	64 '@' $40	65 'A' $41	66 'B' $42
67 'C' $43	68 'D' $44	69 'E' $45	70 'F' $46	71 'G' $47
72 'H' $48	73 'I' $49	74 'J' $4A	75 'K' $4B	76 'L' $4C
77 'M' $4D	78 'N' $4E	79 'O' $4F	80 'P' $50	81 'Q' $51
82 'R' $52	83 'S' $53	84 'T' $54	85 'U' $55	86 'V' $56
87 'W' $57	88 'X' $58	89 'Y' $59	90 'Z' $5A	91 '[' $5B
92 '\' $5C	93 ']' $5D	94 '^' $5E	95 '_' $5F	96 '`' $60
97 'a' $61	98 'b' $62	99 'c' $63	100 'd' $64	101 'e' $65
102 'f' $66	103 'g' $67	104 'h' $68	105 'i' $69	106 'j' $6A
107 'k' $6B	108 'l' $6C	109 'm' $6D	110 'n' $6E	111 'o' $6F
112 'p' $70	113 'q' $71	114 'r' $72	115 's' $73	116 't' $74
117 'u' $75	118 'v' $76	119 'w' $77	120 'x' $78	121 'y' $79
122 'z' $7A	123 '' $7B	124 '|' $7C	125 '' $7D	126 '~' $7E
127 '␡' $7F	128 'Ä' $80	129 'Å' $81	130 'Ç' $82	131 'É' $83
132 'Ñ' $84	133 'Ö' $85	134 'Ü' $86	135 'á' $87	136 'à' $88
137 'â' $89	138 'ä' $8A	139 'ã' $8B	140 'å' $8C	141 'ç' $8D
142 'é' $8E	143 'è' $8F	144 'ê' $90	145 'ë' $91	146 'í' $92
147 'ì' $93	148 'î' $94	149 'ï' $95	150 'ñ' $96	151 'ó' $97
152 'ò' $98	153 'ô' $99	154 'ö' $9A	155 'õ' $9B	156 'ú' $9C
157 'ù' $9D	158 'û' $9E	159 'ü' $9F	160 '†' $A0	161 '°' $A1
162 '¢' $A2	163 '£' $A3	164 '§' $A4	165 '•' $A5	166 '¶' $A6
167 'ß' $A7	168 '®' $A8	169 '©' $A9	170 '™' $AA	171 '´' $AB
172 '¨' $AC	173 '≠' $AD	174 'Æ' $AE	175 'Ø' $AF	176 '∞' $B0
177 '±' $B1	178 '≤' $B2	179 '≥' $B3	180 '¥' $B4	181 'µ' $B5
182 '∂' $B6	183 '∑' $B7	184 '∏' $B8	185 'π' $B9	186 '∫' $BA
187 'ª' $BB	188 'º' $BC	189 'Ω' $BD	190 'æ' $BE	191 'ø' $BF
192 '¿' $C0	193 '¡' $C1	194 '¬' $C2	195 '√' $C3	196 'ƒ' $C4
197 '≈' $C5	198 'Δ' $C6	199 '«' $C7	200 '»' $C8	201 '…' $C9
202 ' ' $CA	203 'À' $CB	204 'Ã' $CC	205 'Õ' $CD	206 'Œ' $CE
207 'œ' $CF	208 '–' $D0	209 '—' $D1	210 '“' $D2	211 '”' $D3
212 '‘' $D4	213 '’' $D5	214 '÷' $D6	215 '◊' $D7	216 'ÿ' $D8
217 'Ÿ' $D9				

This is no complete manual to MacsBug.   This is taken from a file I wrote a while back and is meant to be an extension of the file called "Hex and Such".

--==< The Basics >==--

You install MacsBug simply by throwing it into the System Folder, and by restarting your machine. You activate it by pressing the “command” and “power-key” (the one towards the top of your keyboard marked with the head of an arrow pointing to the left) buttons. This should have “dropped” you into MacsBug. You will notice that you are in MacsBug because your desktop is replaced with a with a bunch of numbers on a white background.
Well, going from the top left side, under “SP” is the current position of the stack pointer, underneath the position of the SP are the values contained in the SP.
Under those numbers is the name of the application that is currently the foremost one.
Under that is the status of the Status Register, followed by the info held in the 8 data registers, and the 8 address registers (or the 32 registers if it is a PPC program). To the right of the registers is a horizontal line going across the screen.
Under that are about 4 lines of text. The topmost line describes where in the application’s code the processor was halted. Under that line are 3 other lines with assembly commands. These are the three commands in line to be executed.
To the right of them (in the right bottom corner) are the hexadecimal values of the assembly commands.
Above this section (in the middle) is a large empty space. This gets filled up with the results of the commands you give to MacsBug.
Your commands get written under the three assembly commands.

What happens when you drop into MacsBug is that your processor stops executing commands, and you can for example go through a code step by step, command by command,
following through exactly what the program does. So how do you do that?
Type “t” followed by a return.
This causes the processor to execute the next assembly command in line. The assembly command that was executed appears now in the middle blank section of your screen, and a new assembly command appears under the two old ones.

--==< Basic MacsBug Commands >==--

- t : traces over the next command in line. If it is used on a JSR command it jumps over the the subroutine. (It executes the whole subroutine, without allowing you to see what happened)

- s : does the same thing as “t” except it “steps into” a subroutine. For example if you are not interested in what happens in an subroutine you should type “t”. This causes the processor to continue until it reaches a RTS command, and only then give the control back to you. If you on the other hand want to see what happens in that subroutine, you should type “s” to step into it and follow through the code from there.

- es : this forces the current application to quit (not always).

- rs : restarts your computer (sometimes it doesn't work and you have to do it the old fashioned way: apple-control-powerkey)

- rb : reboots your computer (boots up the different external devices at startup). This is slower then the “rb” command

- dm [address] : displays what is in the memory at a given address. For example, the command “dm a6” shows you what is held in the address pointed to by address register 6. If you type “dm abcd” it shows you what is held in the memory at location “abcd” (in hex that is).

- db [address] : displays byte from address

- dw [address] : displays word from address

- dl [address] : displays long from address

- il : dissembles the codes. Used if you, for example, want to see what happens after a branch code.

- atb [a-trap name] : MacsBug activates every time that a-trap is being called.

- atc : clears a-traps

- f address expr ‘string’ : this is the find command. “address” refers to the starting point of the search; “expr” is how many bytes it should search; “ ’string' ” is what you’re looking for! Observe the semi quotation mark before the string! You need to use that!

You can find out more about commands for MacsBug by typing “help”