s1x0rteen @ 16 @ sICKSTEEN
@ hex and such @
s1XXT33n @ ProZaq @ sixt33n

Are you a "newbie"? As long as you're interested in not only computers but also in what's making computers work the way they do, then you'll definitely need to learn and master the meaning of a couple of basic expressions/ terms/ concepts. Take, for example, the hexadecimal number system; it doesn't matter if you want to learn the basics of programing or if you want to write programs for Macs or PC's or you just wanna cheat on some computer games; you have to learn and master it in order to be able to "exploit" it. And as you learn more you will notice that all these concepts are interrelated and one can be manipulated to change the other.

In this file I shall try to explain the following topics: binary and hexadecimal numbers, bytes/words/longs, ASCII characters, strings, HexEditors, the hardware components of a computer, and debuggers. If you find that you are not familiar with an expression, then take a look in the "The Computer's Hardware Components" chapter.

--==< Binary, Decimal, and Hexadecimal Numbers >==--

Oh boy! Where do I start? Well, at the very, very, very beginning...
If I remember my IT classes well, the whole fame about binary numbers and calculations with binary numbers goes to an English fellow named George Boole. He developed amongst
others Boolian Algebra. Remember all those horrible hours you had to spend in algebra class learning formulas like: a(b+c) = a*b + a*c? Well you have him to thank for it.
e also developed a type of logic where he used ones and zeros to represent the logical flow of an operation, which is the kind of logic that every personal computer chip uses today.

You know how everyone is always saying that computers are all about ones and zeros? Well that's because everything in computers narrows down to being a one or a zero (an electronic current or the lack of it).

But what on earth is the binary number system? Well, let's try to define the decimal number system first (the one we use in every day mathematics) since we're more familiar with it.

The decimal number system is based on the number 10. Twas the name "Decimal"; which means "tenth" in Latin (doesn't "mal" mean "multiply" in German?). You have the numbers zero through nine. When you start counting from zero up, you hit nine. And what happens when you hit ten? You reset the value of the rightmost column (set it to zero), and carry a one into the next column. At one hundred you reset the two rightmost columns and carry a one into the next one. And so on. So as you notice you carry numbers at the powers of ten. Like 10^1 =10 (^ means raised to the power), 10^2 = 100, 10^3 = 1000, 10^4
= 10 000, 10^5 = 100 000, 10^6 =1 000 000 etc.

Let's break the number "9876" into columns representing the numbers at which the carrying occurs. The "thousands", "hundreds", "tens", and "ones" column.

Thousands	Hundreds	Tens	Ones
(10^4)	(10^3)	(10^2)	(10^1)
9	8	7	6

As you might have noticed, in order to get the number nine thousand eight hundred and seventy six you multiply the value of each column with the appropriate multiple of ten then add the values together (9*10^4 + 8*10^3 + 7*10^2 + 6*10^1).

In the binary number system we only have two numbers to work with instead of ten as we had in decimal. One and zero. So this means, that instead of carrying numbers at the powers of ten we carry numbers at the powers of two; namely: 2^1 = 2, 2^2 = 4, 2^3 = 8, 2^4 = 16, 2^5 = 32, 2^6 = 64, 2^7 = 126 and 2^8 = 256.

When dealing with binary a lot of times the value of all eight columns of numbers are shown even if it is zero. Makes the calculations easier. For example, one in binary has the value 1 but can also be written as 00000001.

Here is a little chart showing the numbers one to sixteen in binary:

Value of column:

126	64	32	16	8	4	2	1	= value of each column added up
0	0	0	0	0	0	0	0	= 0
0	0	0	0	0	0	0	1	= 1
0	0	0	0	0	0	1	0	= 2
0	0	0	0	0	0	1	1	= 3
0	0	0	0	0	1	0	0	= 4
0	0	0	0	0	1	0	1	= 5
0	0	0	0	0	1	1	0	= 6
0	0	0	0	0	1	1	1	= 7
0	0	0	0	1	0	0	0	= 8
0	0	0	0	1	0	0	1	= 9
0	0	0	0	1	0	1	0	= 10
0	0	0	0	1	0	1	1	= 11
0	0	0	0	1	1	0	0	= 12
0	0	0	0	1	1	0	1	= 13
0	0	0	0	1	1	1	0	= 14
0	0	0	0	1	1	1	1	= 15
0	0	0	1	0	0	0	0	= 16

Here's an other approach in trying to explain how binary works. Try adding up the values of the columns where there is a one. In ten for example (00001010) there is a one in the two's and the eight's column. Thus when these values are added together (two plus eight) we get ten. The same goes for fifteen, there's a one in each column so, eight plus four, plus two, plus one equals fifteen.

OK, now we've reached the hexadecimal numbers. Well, for these suckers we carry at powers
of sixteen. With other words we count from zero to fifteen before reseting the first column and increasing the next. The slight problem of only having ten numbers in our everyday number system is compensated by using six alphabetical letters to represent the numbers ten through fifteen. Thus the numbers used in the hexadecimal number system have
the following notation:
0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E, F.

Once fifteen is reached, the next number (as always) is represented by reseting the first column and increasing the next. Meaning that sixteen in hex is "10".

If you have managed to get this far you've done a good job. And if you still have difficulties understanding what the different number systems are all about then I'll let you in on a big secret. Only a very few people convert between number systems in their head. Most of us mortals rely on something called the "Scientific calculator". This makes life a lot simpler! I always use a calculator simply because it's just so much faster. I believe that if you know the principles behind the different number systems and you have access to a calculator that converts between these then you're set.

So now you know what different number systems are. But when it comes to writing them down
some difficulties may arise. It's obviously easy to distinguish numbers represented in binary. Just to be on the safe side, however, it's a convention to put a "%" sign in front of binary numbers. On the other hand "123" can be a number represented in both hex and decimal form. If it's a decimal number it's simply one hundred twenty three. But if it's a hexadecimal number then it has the decimal value of 291, two hundred ninety one. Big difference there! So how do you distinguish between hex and decimal numbers? Well the most common way is to represent hex numbers by putting a dollar sign, "$" in front of the number. In the programing language C you represent decimal numbers using the "0x" prefix. In assembly language it is common practice to use the "#" sign when representing decimal numbers. I tend to be very lazy so when I want to represent decimal numbers I just don't bother using any signs, but for hex numbers I always use the "$" sign. For example: #12345 (decimal) is $3039 (hexadecimal); and $ABCDEF (hexadecimal) is 11259375
(still decimal if no sign is used). Through the course of this file I will use this method of notation. I might, however, refer to hexadecimal numbers without the $ sign if I think that it's obvious what I mean.

--==< Bytes, words, and longs >==--

Now that you know what hex is, there is a need to discuss the length of a number. The length of numbers have a large part when it comes to writing programs. By using numbers with different lengths the programmer can manipulate data much more easily. Another benefit of numbers with different lengths is that a small numbers will occupy a small place in the memory instead of occupying an unnecessarily large one. This is not much of a problem now with the increase of of both RAM and HardDisk sizes, but back in the days of C-64's and before, when programmers only had so much RAM to work with, it was very important wether a number took up 1 or 4 bytes.

Anyway, in assembly language for the 68k Macintosh processors we talk about bytes, words and longs. A byte is two digits long and is between 00 and FF (0 to 255 in dec). A word is 4 digits long and is between 00 00 and FF FF (0 to 65535 in dec). Finally a long is made up of 8 digits and is between 00 00 00 00 and FF FF FF FF (0 to 4294967295 dec).

With other words:

byte:	$00	- $FF	#0 - #255
word:	$00 00	- $FF FF	#0 - #65535
long:	$00 00 00 00	- $FF FF FF FF	#0 - #4294967295

As you can see a byte takes up one fourth of the memory a long does. This principle will be discussed further in the chapter dealing with HexEditors. I think it might be a good thing for you to learn how many digits a byte, a word and a long has. I will use these expressions later on. I chose to use these expressions (and not including floats and doubles) because I feel that even an experienced person can get far with only these three length-notations.

For those interested, the programing language C uses the following expressions to refer to the length of numbers:

```
char                c = 'A';   // 1-byte long by definition (in C++).
short int           si= 1;     // minimum range +/-32767.
short               s = 2;     // short same as short int.
int                 i = 3;     // minimum range +/-32767.
long int            li= 4;     // minimum range +/-2147483647.
long                l = 5;     // long same as long int.
float               f = 10.1; // min 6 digits (decimal) precision.
double              d = 11.2; // min 10 digits (decimal) precision.
long double         ld= 12.3;

unsigned char       uc;        // unsigned integers can only store
unsigned short int  usi;       // positive numbers.
unsigned int        ui;
unsigned long int   uli;

signed char         sc;        // signed integers can store positive
signed short int    ssi;       // or negative numbers.
signed int          si2;
signed long int     sli;
```

(Information taken from "C Reference Card" by Argus Software Engineering)

--==< ASCII Characters >==--

With the arrival of networks reaching from one country to the other arose the problem of character mapping. When you push the letter "a" on your keyboard, the hardware components
of the computer send a number value to the processor which represents the letter "a". But how on earth would a computer in Yugoslavia, configured to deal with the Yugoslavian alphabet, be able to interpret letter "ä" which is fairly common in the Swedish language. To eliminate the problem a new standard for keyboards, the American Standard Code for Information Interchange (ASCII) was adopted in most places. What this means is that (in theory at least) all alphabetical characters will appear the same way no matter where you are in the world. Unfortunately this only works in theory, since different keyboards have different mapping of different keys and have different ways of showing different letters etc... The good news is that just like you didn't have to know how to convert hex numbers in your head, it's enough that you know that ASCII refers to the numerical values of the different characters on your keyboard that the computer can interpret as such.

Now you know that when you push a key on the keyboard, the corresponding number value is
sent to the processor (well in reality it's interpreted by the OS and sent to the active application). So, what is this number value? Well, every character on the keyboard is

represented by a different number. For example the English lowercase alphabetical characters range from $61 to $7A (a-z). Notice that when it comes to computers there's a define difference between lowercase and uppercase letters. Thus the uppercase English letters are represented by the numbers $41 to $5A (A-Z).

It is important to realize that every ASCII character (every character on the keyboard) can be represented by a number that's the size of a byte. Meaning a number between 1-255,
$1-FF. Thus the current standard of keyboard maps can only handle 255 characters.

But that's of no real importance either. The most common ASCII characters and their values in both hex and decimal form are available in the included file "ASCII.txt"

Now then, we know that ASCII characters are represented by numbers. For example the capital letter "A" is represented by 65 ($41). "B" is 66 ($42) and "C" is 67 ($43). So the letters "ABC" could be represented by the ASCII values 65 66 67 (or in hex 41 42 43). And this brings us to our next topic, strings.

--==< Strings >==--

The expression "string" refers to a sequence of keyboard characters. For example "Hello world!" would be a string. Notice that the computer doesn't care about the space between the two words, it looks upon the sentence as only one string of characters. This leads to the problem of representing strings. Imagine how a string would look like in the computer's point of view. It would be a sequence of numbers stored somewhere in the memory. And unless you inform the computer how to interpret the beginning or end of the string, it will not know where the string ends.

There are currently two standard ways of representing strings. The C way and the Pascal way. I'll start with the C way, it's easier. Basically after the last character in the string there is a zero-byte. This means that a value of zero marks the end of the string. For example:

H	E	L	L	O	_	W	O	R	L	D	!	•
72	69	76	76	79	95	87	79	82	76	68	33	00
$48	$45	$4c	$4c	$4f	$5f	$57	$4f	$52	$4c	$44	$21	$00

Keeping in mind that the size of an ASCII character is that of a byte (max 255) we notice that using the C method the length of the string is actual increased by one byte; the zero-byte on the end. When a program is in need of using the above string, it needs to know the memory address of the first character, and it knows that it has hit the end of the string when the value of the character is zero.

The Pascal method is a bit different. It stores the number of characters in the string as the first byte. The example above would be portrayed like this in Pascal notation:

•	H	E	L	L	O	_	W	O	R	L	D	!
12	72	69	76	76	79	95	87	79	82	76	68	33
$0c	$48	$45	$4c	$4c	$4f	$5f	$57	$4f	$52	$4c	$44	$21

As you might have noticed there are 12 characters in the string (including the "_" and

the "!" signs). So using the Pascal method, the program would read the first byte of the string and thus determine the lenght of it.

This whole concept will be developed further in the next chapter.

--==< Hex Editors >==--

NOTICE: When dealing with hex editors you are going to be changing real files on your computer. By changing just one byte in a file you can corrupt it to the extent that it will not be usable any more! So always make sure that you are working on a BACKUP of the file. The easiest thing to do is to create a folder where you copy all the files that you want to change with the HexEditor.

Remember how all data processed by the computer is made up of a one or a zero? Well, the
same principle holds true for files stored on the hard disk, on a floppy disk, on a CD-ROM, or on any other storage media. But because hexadecimal numbers are easier to deal with than binary numbers, we have programs that can read the content of any storage media as pure hexadecimal data. These programs are called HexEditors. Using the above idea, any file containing data that is stored on a media can be opened and it's contents will be represented as hexadecimal numbers. And it does not matter whether the file is an application program or just a simple text file, since ALL files are at their "lowest level" made up of binary numbers and can thus be viewed by a HexEditor.

The first thing you have to do is to find yourself a HexEditing program. It doesn't matter which computer platform you have. HexEditors exists for PC's, Mac's, Unix's, even C-64's Once you've found a HexEditor open up any backup file with the program. I have a Mac and I
use HexEdit 1.0.7, a freeware program by Jim Bumgardner. If I open an application file I get something like this:

(See the picture "HexEdit.jpg")

Please note that you WILL get something completely different, since the chances of us opening the same file is very slim, and different HexEditors present the information in different ways.

Let me explain the above picture. To the left you have the Offset column. "Offset" refers to the distance of a data from the first byte in the file. Since the offset here starts at zero we know that we are dealing with the beginning of the file. Also notice that the offsets are displayed as hex values. A good HexEditor should be able to display the offset as decimal numbers as well.

In the middle you have the Hex column. This is where all the hexadecimal data can be found. If you converted all these numbers to binary, you'd have a representation of the binary information of the file as you would find it on the Hard Drive.

Finally on the right side is the ASCII column. This is an ASCII representation of the Hex values. This means that each hex number is looked up on an ASCII table and it's ASCII value is displayed in this column.

OK, now what? Well, as an example I'll describe the use of HexEditors as a way to cheat

on computer games.

Off course you can not use a HexEditor to cheat on a game while you are playing it. Those situations will be dealt with in the next chapter. What you can do with a HexEditor, however, is to change saved games. I mean, think about it. What is the program actually doing when it is saving a game? It saves all the data about the game to a file. Like where you are positioned on the map, what items you carry, how many monsters are gonna attack you etc... In this example I will use Realmz, a shareware game for the MacOS.

The first thing you have to do is to find where on the HardDrive the game saves it's files. Some games allow you to save wherever you want, while others will only allow you to save into a certain set of game folders (usually 1-10 or something like that). So search through the game's folders (directories as they are also called), and look for a file that has the same name as your saved game.

The next step is to find the document in which the game stores the information you want to change. For example Realmz is a Dungeons & Dragons game for the Mac where you can create your own characters. The attributes of the characters, such as it's strength or stamina, are saved in a file that has the same name as the character.

Let us presume that I have a character called Pro. His attributes are stored in the file called "Pro". I want to change my character's strength. I want to make him stronger so that he can cause more damage with each hit. The first thing I would do is to run the game and see how strong he is at that particular time. This will be the value that the game stores in the "Pro" file. He has a strength of 105. So I convert this number to hex, which gives me $69. And then I set out to look for the hex byte $69 in the saved file. To make things easier I look for the hex word "00 69" since the possibility of the string "00 69" appearing several times in the file is smaller than that of the string "69". (Read "Note on HexEditors and numbers" for more information regarding this.) When I've found this value I change it to whatever I want it to be and then I save my work.

The problem might arise that "00 69" appears in more than one places in the file. The easiest (and most dangerous way) is to change all the values to the value you want. By doing this, however, you might have changed values which are very important to the program
and might cause it to freeze. By using a trial end error method you can try to change a different value every time and see if the value you changed was the correct one. The most effective method, however, is to look at "00 69" in a context. Meaning, look at the other numbers around it. For instance, if you recognize the number after "00 69" as the movement points of the character then there's a good chance that you're on the right track.

Note for Macintosh users: The MacOS divides up a file into two parts, the data fork and the resource fork. Without getting too much into programing, here's what the purpose of these two forks are. The resource fork should contain information such as how a window looks like, where it is located, how the menus look like etc. With other words information used by the Operating System. The data fork should be used to store the information used by the user's. For example, in a word processor file the resource fork might contain information regarding the size of the window, while the data fork might contain the actual text written by the user. However, the programmer is not obliged to follow these criterias. They are only suggestions made by Apple. So, when you are looking at a file with a HexEditor on a Mac, be sure to check both forks of the file for the information you are looking for.

--==< Note On HexEditors And Numbers >==--

I find it appropriate to give a bit of a revision of numbers and strings.
To use the example from above, let's presume that my character had the strength of $69. What we don't know is how the program stores this number. It might store it as a byte, a word, or a long (see chapter about bytes, words and longs for more info about this). Using common sense, if my character has a strength of $69 and is considered very very strong than the program will probably save the value as a byte or a word. It's completely useless for it to store it as a long (although it might happen). If, however, we regard the characters experience point, its obvious that it is a lot larger than the range of a word, so it HAS to be stored in a long (or something larger). So instead of searching for "ABCDE" you can search for "00 0A BC DE" which should narrow down the number of occurrences of that number.

Another thing that needs to be discussed is that the length of a number has to be even. A programmer deals with blocks (units) of memory. The program then reserves these blocks
once it's launched. The smallest block a programmer deals with is a byte. This means that no matter how much the programmer wants it, he/she can never store the number "1" just like that. If it is to be stored in the memory it will be stored as "01". However, if the programmer assigned the number to be a word it will be stored as "00 01". And if it was assigned to be a long it will be stored as "00 00 00 01". The computer doesn't care what number is stored in the variable. It only cares about the length of the variable. Thus if the computer stores three longs with the values $1, $22 and $333 respectively then it will look like this once you open the file with a HexEditor:

00 00 00 01 00 00 00 22 00 00 03 33

Lets say you want to change the $333 part to $433. A good HexEditor might allow you to search for "333" but remember that the smallest unit is a byte. When you are changing "00 00 03 33" to "00 00 04 33" it's pointless to change all 8 digits. It's enough if you change the 3'rd byte ("03" to "04"). Notice, however, that you can't just change 3 to 4. You have to change "03" to "04". A good HexEditor should actually not allow you to change one digit at a time. It should require you to change one byte, 2 digits, at a time. If you are confused then re-read this chapter, and the previous chapter dealing with lengths of numbers. This is important stuff, and it's very important that you know it well!