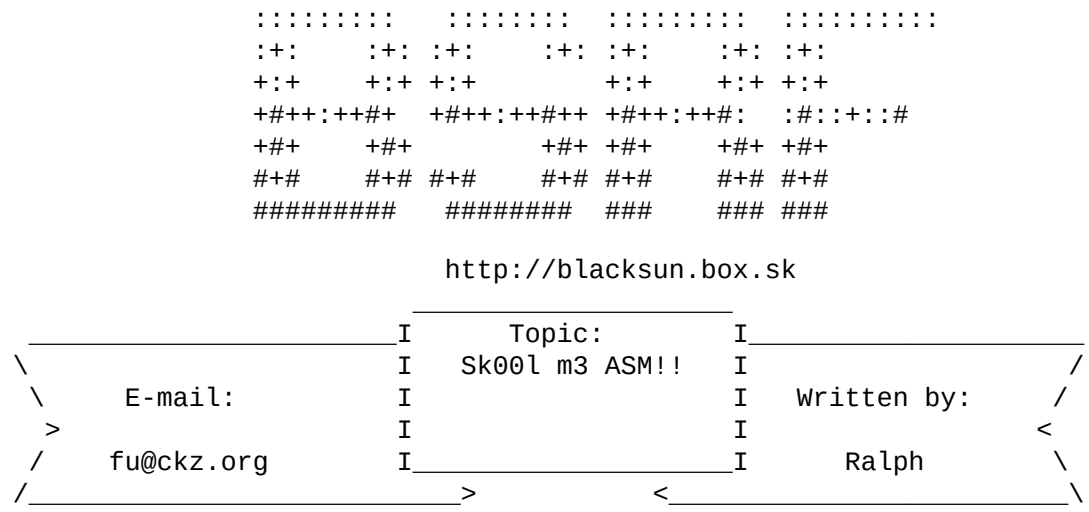


Sk00l m3 ASM!!#@$!@#

by Ralph (fu@ckz.org)
-AWC (http://awc.rejects.net)
Version: 0.841
Date: 7/23/00

NOTE: This thing is almost done, just gotta finish of the Win32 section, however I started working on other shit so finishing this is kinda 10th on my priority list. If you think you can convince me to finish it sooner, feel free to contact me.

TOC

1. Introduction
-What is it?
-Why learn it?
-What will this tutorial teach you?

2. Memory
-Number Systems
-Decimal
-Binary
-Hexadecimal
-Bits, Nybbles, Bytes, Words, Double Words
-The Stack
-Segment:Offset
-Registers

3. Getting started
-Getting an assembler
-Program layout
-.COM
-.EXE

4. Basic ASM
-Basic Register operations
-Stack operations
-Arithmetic operations
-Bit wise operation
-Interrupts

5. Tools

    -Debug
    -CodeView

6. More basics
    -.COM file format
    -Flow control operations
    -Loops
    -Variables
    -Arrays
    -String Operations
    -Sub-Procedures
    -User Input

7. Basics of Graphics
    -Using interrupts
    -Writing directly to the VRAM
    -A line drawing program

8. Basics of File Operations
    -File Handles
    -Reading files
    -Creating files
    -Search operations

9. Basics of Win32
    -Introduction
    -Tools
    -A Message Box
    -A Window

Appendix A
    -Resources

Appendix B
    -Credits, Contact information, Other shit

1. Introduction

What is it?

Assembly language is a low-level programming language. The syntax is nothing like C/C++, Pascal, Basic, or anything else you might be used to.

Why learn it?

If you ask someone these days what the advantage of assembly is, they will tell you it's speed. That might have been true in the days of BASIC or Pascal, but today a C/C++ program compiled with an optimized compiler is as fast, or even faster than the same algorithm in assembly. According to many people assembly is dead. So why bother learning it?

1. Learning assembly will help you better understand just how a computer works.
2. If windows crashes, it usually returns the location/action that caused the error. However, it doesn't return it in C/C++. Knowing assembly is the only way to track down bugs/exploits and fix them.
3. How often do you whish you could just get rid of that stupid nag screen in that

shareware app you use? Knowing a high-level language wont get you very far when you open the shit up in your decompiler and see something like CMP EAX, 7C0A
4. Certain low level and hardware situations still require assembly
5. If you need precise control over what your program is doing, a high level language is seldom powerful enough.
6. Anyway you put it, even the most optimized high level language compiler is still just a general compiler, thus the code it produces is also general/slow code. If you have a specific task, it will run faster in optimized assembly than in any other language.
7. "Professional Assembly Programmer" looks damn good on a resume.

My personal reason why I think assembly is the best language is the fact that you're in control. Yes all you C/C++/Pascal/Perl/etc coders out there, in all your fancy high level languages you're still the passenger. The compiler and the language itself limit you. In assembly you're only limited by the hardware you own. You control the CPU and memory, not the otherway around.

What will this tutorial teach you?

I tryed to make this an introduction to assembly, so I'm starting from the beginning. After you've read this you should know enough about assembly to develop graphics routines, make something like a simple database application, accept user input, make Win32 GUIs, use organized and reuseable code, know about different data types and how to use them, some basic I/O shit, etc.

2. Memory

In this chapter I will ask you to take a whole new look at computers. To many they are just boxes that allow you to get on the net, play games, etc. Forget all that today and think of them as what they really are, Big Calculators. All a computer does is Bit Manipulation. That is, it can turn certain bits on and off. A computer can't even do all arithmetic operations. All it can do is add. Subtraction is achieved by adding negative numbers, multiplication is repeaded adding, and dividing is repeaded adding of negative numbers.

Number systems

All of you are familiar with at least one number system, Decimal. In this chapter I will introduce you to 2 more, Binary and Hexadecimal.

Decimal

Before we get into the other 2 systems, lets review the decimal system. The decimal system is a base 10 system, meaning that it consists of 10 numbers that are used to make up all other number. These 10 numbers are 0-9. Lets use the number 125 as an example:

	Hundreds	Tens	Units
Digit	1	2	5
Meaning	1x10^2	2x10^1	5x10^0
Value	100	20	5

NOTE: x^y means x to the power of y. ex. 13^3 means 13 to the power of 3 (2197)
Add the values up and you get 125.

Make sure you understand all this before going on to the binary system!

Binary

The binary systems looks harder than decimal at first, but is infact quite a bit easier since it's only base 2 (0-1). Remember that in decimal you go "value x 10^position" to get the real number, well in binary you go "value x 2^position" to get the answer. Sounds more complicated than it is. To better understand this, lets to some converting.

Take the binary number 10110:
1 x 2^4 = 16
0 x 2^3 = 0
1 x 2^2 = 4
1 x 2^1 = 2
0 x 2^0 = 0
Answer: 22

NOTE: for the next example I already converted the Ax2^B stuff to the real value:
      2^0 = 1
      2^1 = 2
      2^2 = 4
      2^3 = 8
      2^4 = 16
      2^5 = 32
      etc....

Lets use 111101:
1 x 32 = 32
1 x 16 = 16
1 x 8 = 8
1 x 4 = 4
0 x 2 = 0
1 x 1 = 1
Answer: 61

Make up some binary numbers and convert them to decimal to practise this. It is very important that you completely understand this concept. If you don't, check Appendix B for links and read up on this topic BEFORE going on!

Now lets convert decimal to binary, take a look at the example below:
238 / 2 remainder: 0
119 / 2 remainder: 1
59 / 2 remainder: 1
29 / 2 remainder: 1
14 / 2 remainder: 0
7   / 2 remainder: 1
3   / 2 remainder: 1
1   / 2 remainder: 1
0   / 2 remainder: 0
Answer: 11101110

Lets go through this:
1. Divide the original number by 2, if it divides evenly the remainder is 0
2. Divide the answer from the previous calculation (119) by 2. If it wont
    divide evenly the remainder is 1.
3. Round the number from the previous calculation DOWN (59), and divide it by 2.
    Answer: 29, remainder: 1
4. Repeat until you get to 0....
The final answer should be 011101110, notice how the answer given is missing the 1st 0?
That's because just like in decimal, they have no value and can be omitted (023 = 23).

Practise this with some other decimal numbers, and check it by converting your answer back to binary. Again make sure you get this before going on!

A few additional things about binary:
* Usually 1 represents TRUE, and 0 FALSE
* When writing binary, keep the number in multiples of 4
  ex. DON'T write 11001, change it to 00011001, remember that the 0 in front
  are not worth anything
* Usually you add a b after the number to signal the fact that it is a binary number

ex. 00011001 = 00011001b

Hexadecimal
Some of you may have notice some consistency in things like RAM for example. They seem to always be a multiple of 4. For example, it is common to have 128 megs of RAM, but you wont find 127 anywhere. That's because computer like to use multiples of 2, 4, 8, 16, 32, 64 etc. That's where hexadecimal comes in. Since hexadecimal is base 16, it is perfect for computers. If you understood the binary section earlier, you should have no problems with this one. Look at the table below, and try to memorize it. It's not as hard as it looks.

Hexadecimal	Decimal	Binary
0h	0	0000b
1h	1	0001b
2h	2	0010b
3h	3	0011b
4h	4	0100b
5h	5	0101b
6h	6	0110b
7h	7	0111b
8h	8	1000b
9h	9	1001b
Ah	10	1010b
Bh	11	1011b
Ch	12	1100b
Dh	13	1101b
Eh	14	1110b
Fh	15	1111b

NOTE: the h after each hexadecimal number stands for <insert guess here>

Now lets do some converting:
Hexadecimal to Decimal
2A4F

F x 16^0 = 15 x 1 = 15
4 x 16^1 = 4 x 16 = 64
A x 16^2 = 10 x 256 = 2560
2 x 16^3 = 2 x 4096 = 8192
Answer: 10831

1. Write down the hexadecimal number starting from the last digit
2. Change each hexadecimal number to decimal and times them by 16^postion
3. Add all final numbers up

Confused? Lets do another example: DEAD
D x 1 = 13 x 1 = 13
A x 16 = 10 x 16 = 160
E x 256 = 14 x 256 = 3584
D x 4096 = 13 x 4096 = 53248
Answer: 57005

Practise this method until you get it, then move on.

Decimal to Hexadecimal
Study the following example:
1324

1324 / 16 = 82.75
82 x 16 = 1312
1324 - 1312 = 12, converted to Hexadecimal: C

```
82 / 16 =  5.125
 5 x 16 = 80
82 - 80 =  2, converted to Hexadecimal: 2

5 / 16 = 0.3125
0 x 16 = 0
5 -  0 = 5, converted to Hexadecimal: 5
```

Answer: 52C

I'd do another example, but it's too much of a pain in the ass, maybe some other time.

Learn this section you WILL need it!
This was already one of the hardest parts, the next sections should be a bit easier

Some additional things abot hexidecimal
1. It's not uncommon to say "hex" instead of "hexidecimal" even thechnicaly speaking "hex" means 6, not 16.
2. Keep hexidecimal numbers in multiples of 4, adding zeros as necessary
3. Most assemblers can't handle numbers that start with a "letter" because they don't know if you mean a label, instruction, etc. In that case there are a number of other ways you can express the number. The most common are:
   DEAD = 0DEADh (Usually used for DOS/Win)
   and
   DEAD = 0xDEAD (Usually used for *Nix based systems)
   Consult your assembler's manual to see what it uses.

By the way, does anyone think I should add Octal to this...?

Bits, Nibbles, Bytes, Words, Double Words

Bits are the smallest unit of data on a computer. Each bit can only represent 2 numbers, 1 and 0. Bits are fairly useless because they're so damn small so we got the nibble. A nibble is a collection of 4 bits. That might not seem very interesting, but remember how all 16 hexadecimal numbers can be represented with a set of 4 binary numbers? That's pretty much all a nibble is good for.

The most important data structure used by your computer is a Byte. A byte is the smallest unit that can be accessed by your processor. It is made up of 8 bits, or 2 nibbles. Everything you store on your hard drive, send with your modem, etc is in bytes. For example, lets say you store the number 170 on your hard drive, it would look like this:

```
+---+---+---+---+---+---+---+---+
| 1 | 0 | 1 | 0 | 1 | 0 | 1 | 0 |
+---+---+---+---+---+---+---+---+
  7   6   5   4   3   2   1   0
  H.O Nibble    |   L.O Nibble
```

10101010 is 170 in binary. Since we can fit 2 nibbles in a byte, we can also refer to bits 0-3 as the Low Order Nibble, and 4-7 as the High Order Nibble
Next we got Words. A word is simply 2 bytes, or 16 bits. Say you store 43690, it would look like this:

```
+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+
| 1 | 0 | 1 | 0 | 1 | 0 | 1 | 0 | 1 | 0 | 1 | 0 | 1 | 0 | 1 | 0 |
+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+
 15  14  13  12  11  10   9   8   7   6   5   4   3   2   1   0
       High Order Byte          |         Low Order Byte
```

Again, we can refer to bits 0-7 as Low Order Byte, and 7-15 as High Order Byte.

Lastly we have a Double Word, which is exactly what it says, 2 word, 4 bytes, 8 nibbles or 32 bits.

NOTE: Originally a Word was the size of the BUS from the CPU to the RAM. Today most computers have at least a 32bit bus, but most people were used to 1 word = 16 bits so they decided to keep it that way.

The Stack

You have probably heard about the stack very often. If you still don't know what it means, read on. The stack is a very useful Data Structure (anything that holds data). Think of it as a stack of books. You put one on top of it, and that one will be the first one to come of next. Putting stuff on the stack is called Pushing, getting stuff from the stack is called Poping. For example, say you have 5 books called A, B, C, D, and E stack on top of each other like this:

A
B
C
D
E

Now you add (push) book F to the stack:

F
A
B
C
D
E

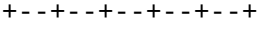
If you pop the stack, you get book F back and the stack looks like this again:

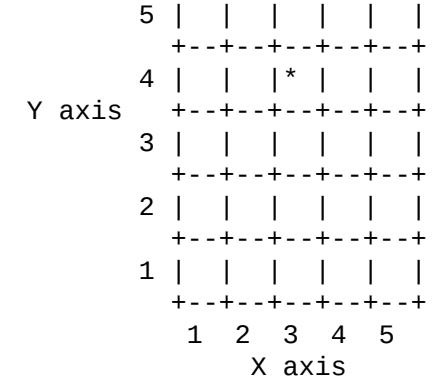
A
B
C
D
E

This called LIFO, Last In, First Out.
So what good is all this? The stack is extremely useful as a "scratchpad" to temporarily hold data.

Segment:Offset

Everything on your computer is connected through a series of wires called the BUS. The BUS to the RAM is 16 bits. So when the processor needs to write to the RAM, it does so by sending the 16 bit location through the bus. In the old days this meant that computers could only have 65535 bytes of memory (16 bits = 1111111111111111 = 65535). That was plenty back than, but today that's not quite enough. So designers came up with a way to send 20 bits over the bus, thus allowing for a total of 1 MB of memory
In this new design, memory is segmented into a collection of bytes called Segments, and can be access by specifying the Offset number within those segments. So if the processor wants to access data it first sends the Segment number, followed by the Offset number. For example, the processor sends a request of 1234:4321, the RAM would send back the 4321st byte in segment number 1234.
This all might sound a bit complicated, but study it carefully and you should be able to master segment:offset.
The best way to picture seg:off is with a 2 dimensional array. Remember that X,Y shit you had to learn in grade 9 math?
Look at the diagram below, the * is located at 4:3. The Y-axis is equal to the segment, and the X-axis is the offset.





To get the physical address do this calculation:
Segment x 10h + Offset = physical address
For example, say you have 1000:1234 to get the physical address you do:
1000 X 10h = 10000

```
 10000
+ 1234
------
 11234
```

This method is fairly easy, but also fairly obsolete. Starting from the 286 you can work in Protected Mode. In this mode the CPU uses a Look Up Table to compute the seg:off location. That doesn't mean that you cannot use seg x 10h + off though, you will only be limited to working in Real Mode and your programs can't access more than 1 MB. However by the time you know enough to write a program even close to this limit, you already know how to use other methods (for those comming from a 50 gig hard drive world, a program that's 1 MB is about 15x bigger than this text file is).

Registers

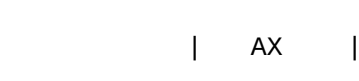
A processor contains small areas that can store data. They are too small to store files, instead they are used to store information while the program is running.
The most common ones are listed below:

General Purpose:
NOTE: All general purpose registers are 16 bit and can be broken up into two 8 bit registers. For example, AX can be broken up into AL and AH. L stands for Low and H for High. If you assign a value to AX, AH will contain the first part of that value, and AL the last. For example, if you assign the value DEAD to AX, AH will contain DE and AL contains AD. Likewise the other way around, if you assign DE to AH and AD to AL, AX will contain DEAD

AX - Accumulator.
     Made up of: AH, AL
     Common uses: Math operations, I/O operations, INT 21
BX - Base
     Made up of: BH, BL
     Common uses: Base or Pointer
CX - Counter
     Made up of: CH, CL
     Common uses: Loops and Repeats
DX - Displacement
     Made up of: DH, DL
     Common uses: Various data, character output

When the 386 came out it added 4 new registers to that category: EAX, EBX, ECX, and EDX.
The E stands for Extended, and that's just what they are, 32bit extensions to the originals. Take a look at this diagram to better understand how this works:

```
|        EAX        |
```



Each box represents 8 bits
NOTE: There is no EAH or EAL

Segment Registers:
NOTE: It is dangerous to play around with these!
CS - Code Segment. The memory block that stores code
DS - Data Segment. The memory block that stores data
ES - Extra Segment. Commonly used for video stuff
SS - Stack Segment. Register used by the processor to store return addresses from routines

Index Registers:
SI - Source Index. Used to specify the source of a string/array
DI - Destination Index. Used to specify the destination of a string/array
IP - Instruction Pointer. Can't be changed directly as it stores the address of the next instruction.

Stack Registers:
BP - Base pointer. Used in conjunction with SP for stack operations
SP - Stack Pointer.

Special Purpose Registers:
IP - Instruction Pointer. Holds the offset of the instruction being executed
Flags - These are a bit different from all other registers. A flag register is only 1 bit in size. It's either 1 (true), or 0 (false). There are a number of flag registers including the Carry flag, Overflow flag, Parity flag, Direction flag, and more. You don't assign numbers to these manually. The value automatically set depending on the previous instruction. One common use for them is for branching. For example, say you compare the value in BX with the value in CX, if it's the same the flag would be set to 1 (true) and you could use that information to branch of into another area of your program.

There are a few more registers, but you will most likely never use them anyway.

Exercises:
1. Write down all general purpose registers and memorize them
2. Make up random numbers and manually convert them into Binary and hexadecimal
3. Make a 2D graph of the memory located at 0106:0100
4. Get the physical address of 107A:0100

3. Getting Started

Now finally on to real assembler! Believe me, I'm getting sick of all this background shit :)

Getting an Assembler

There are quite a few available these days. All code in this tutorial has been tested with TASM, so you should have no problems if you have it. A86 should also work with thise code, but I can't guarentee that.
A86 - Available from: http://eji.com/a86/index.htm
      License: Shareware

      Price: A86 only - US$50+Tax/SH
             A86 + Manual + D86 + 32bit version of each - US$80+Tax/SH
             Manual - US$10+Tax/SH
TASM - Available from: http://www.borland.com/borlandcpp/cppcomp/tasmfact.html
       License: Gotta buy it
       Price: US$129.95+Tax/SH
There are tons more out there, check www.tucows.com, I know they have a few. However as said before, all programs in this tutorial have only been tested with TASM. If you are low on cash, just get A86 and evaluate for longer than you're supposed to.

Program Layout

It is good programming practise to develop some sort of standard by which you write your programs. In this chapter you will learn about a layout of .COM and .EXE files that is excepted by both, TASM and A86.

.COM

Lets look at the source code to a very simple program:

```
MAIN  SEGMENT
      ASSUME CS:MAIN,DS:MAIN,ES:MAIN,SS:MAIN

      ORG 100h

START:
  INT 20
MAIN ENDS
END START
```

This program does absolutely nothing, but it does it well and fast. Lots of code for something that doesn't do shit. Lets examine that more closely:

MAIN SEGMENT - Declares a segment called MAIN. A .COM file must fit into 1 segment of memory (aka. 65535 bytes - stack - PSP)

ASSUME CS:MAIN,DS:MAIN,ES:MAIN,SS:MAIN - This tells the assembler the initial values of the CS,DS,ES, and SS register. Since a .COM must fit into one segment they all point to the segment defined in the line above.

ORG 100h - Since all .COM files start at XXXX:0100 you declare the entry point to be 100h. More on this later.

START: - A label

INT 20 - Returns to DOS

MAIN ENDS - Ends the MAIN segment

END START - Ends the START label

NOTE: This is the kind of layout we will use most of the time. Maybe later (Chapter 9+) We get into the next one...

Now how do you make this shit into an actual program? First, type this program out in your favourite editor (notepad, dos edit, etc). If you have A86, just get into DOS, than into the directory A86 is in and type "a86 filename.asm". If you have TASM, get into DOS and into the tasm directory and type "tasm filename.asm", then type "tlink /t filename.obj". In both cases you will get a file called Filename.com. More on what .com is and does later.

.EXE
Take a look at the following code:

```
    DOSSEG
    .MODEL SMALL
    .STACK 200h
    .DATA
    .CODE

START:
    INT 20
END START
```

Again, his program does absolutely nothing. Lets examine each line in detail:

DOSSEG - sorts the segments in the order:
         Code
         Data
         Stack
This is not required, but recommended while you're still learning

.MODEL SMALL - selects the SMALL memory model, available models are:
TINY: All code and data combined into one single group called DGROUP. Used for .COM
         files.
SMALL: Code is in a single segment. All data is combined in DGROUP. Code and data
         are both smaller than 64k. This is the standard for standalone assembly
         programs.
MEDIUM: Code uses multiple segments, one per module. All data is combined in DGROUP
         Code can be larger than 64k, but data has to be smaller than 64k
COMPACT: Code is in a single segment. All near data is in DGROUP. Data can be more
         than 64k, but code can't.
LARGE: Code uses multiple segments. All near data is in DGROUP. Data and code can be
         more than 64k, but arrays can't.
HUGE: Code uses multiple segments. All near data is in DGROUP. Data, code and arrays
         can be more than 64k
Most of the time you want to use SMALL to keep the code efficient.

.STACK 200h - sets up the stack size. In this case 200h bytes

.DATA - The data segment. This is where all your data goes (variables for example).

.CODE - The code segment. This is where your actually program goes.

START: - Just a label, more on that later

INT 20 - exits the program, more on that later

END START - Take a wild guess on what this does! Not required for all assemblers, I
            know TASM needs it, and A86 doesn't.

NOTE: For .EXE files, DON'T use the /t switch when linking!

Exercises:
1. Make a program that uses the LARGE memory model, sets up a 100h long stack, and exits
    to DOS.

==========
In this chapter we actually start making some working code

Basic Register operations

You already know what registers are, but you have yet to learn how to modify them.
To assign a value to a register:

```
MOV DESTINATION, VALUE
```

For example, say you want AX to equal 56h:

```
MOV AX,56h
```

You can also use another register as the value:

```
MOV AX,BX
```

Remember how all general purpose registers are made up of a H and a L register? Now you can actually use that info:

```
MOV AL,09
```

Now AL equals 09 and AX equals 0009

The next register operator is XCHG, which simply swaps 2 registers. The syntax is:

```
XCHG REGISTER1, REGISTER2
```

For example, consider the following code:

```
MOV  DX,56h
MOV  AX,3Fh
XCHG DX,AX
```

1. DX is equal to 3Fh
2. AX is equal to 56h
3. DX and AX get swapped and AX now equals 56h, and DX equals 3Fh

NOTE: NEVER try to exchange a 8 bit (h/l) register with a 16 bit (X)!!
The following code is invalid:

```
XCHG AH,BX
```

Next we got 2 simple operations, INC and DEC.
INC increment a register's value and DEC decrements it.

Example:

```
MOV DX,50h
INC DX
```

DX is now equal to 51h (50h + 1h = 51h).

Example:

```
MOV DX,50h
DEC DX
```

DX is now equal to 4F (50h - 1h = 4Fh).

Stack operations

Now it's time to put that stack shit to some actual use. And it's very easy to. There are 6 stack operators, 2 of which you will use most of the time. The syntax is:

```
POP  REGISTER
PUSH REGISTER
```

Lets say you want to temporarily store the value in AX on the stack for later use,

simply do:
PUSH AX
Now you played around with AX and want to restore the original value:
POP AX

NOTE: The stack will only accept 16 bit registers! That shouldn't be a problem though
      since the 16 bit registers include the value of the 8 bit.

This next bit of code does some poping and pushing, take a guess on what BX and AX are
equal to at the end.

```
MOV  AX,51h
MOV  BX,4Fh

XCHG AX,BX

PUSH AX
MOV  AX,34h

POP  BX

PUSH BX

POP  AX
```

First AX is equal to 51h and BX to 4Fh, than the 2 get exchanged. Now we got
AX = 4Fh and BX = 51h. AX gets pushed on the stack, then set to 34h:
AX = 34h and BX = 51h. BX gets poped, than pushed:
AX = 34h and BX = 4Fh. Finally AX gets poped. So the final result is:
AX = 4Fh and BX = 4Fh.

Next we got the two variations of the stacks registers, POPF and PUSHF. These two
place the flag register on the stack. Sounds more complicated than POP and PUSH, but
it's actually easier. The syntax is:
POPF
PUSHF
No operand is required. For example, say you want AX to hold the current flag register
value:

```
PUSHF
POP AX
```

PUSHF puts it on the stack, POP AX places it into AX.

The last two stack operators are PUSHA and POPA.
PUSHA puts all general purpose registers on the stack
POPA retrieves all general purpose registers from the stack
NOTE: These 2 are 32bit instructions, so they only work on a 386+ and will not
       work with .COM files.

Example:

```
MOV AX,1h
MOV BX,2h
MOV CX,3h
MOV DX,4h

PUSHA

MOV AX,5h
MOV BX,6h
MOV CX,7h
MOV DX,8h
```

POPA

At the end of this program, all registers are restored to their initial value

Practise some of these instructions! If you make a program containing everything you've learned so far it won't do anything, but if it doesn't crash it most likely worked. So code some simple programs and play around with the values and registers.

Arithmetic operations

Everyone loves arithmetic. Especially if you do it in hex or binary. For those who don't know what arithmetic is, it's just adding and subtracting. Multiplying and dividing are really just repeated additions and subtractions. So in short, it's a fancy name for grade 3 math. In this chapter I will introduce you to the 4 basic arithmetic operators, ADD, SUB, MUL, DIV . There are a few more that I will cover later.
Lets start with ADD. The syntax is:
ADD REGISTER1, REGISTER2
ADD REGISTER, VALUE

Example 1:
```
MOV AX,5h
MOV BX,4Fh

ADD AX,BX
```

This adds AX and BX and stores the resulting value in AX. So after running this program AX = 9h

Example 2:
```
MOV AX,5h

ADD AX,4Fh
```

The result is the same as in example 1. AX is set to 5h, and 4Fh is added to it.

Now lets go on to SUB. The syntax is:
SUB REGISTER1, REGISTER2
SUB REGISTER, VALUE

Example 1:
```
z
```

This will subtract the value of BX from the value of AX. In this case the result would be 4A.

NOTE: If you still don't completely get hexadecimal, you can easily check this by converting 5, 4F, and 4A to decimal.
4F = 79
4A = 74
 5 = 5

As with ADD you can also use a value:
```
MOV BX,4Fh

SUB BX,5h
```

Which leaves you with BX = 4A

Next in line is the MUL operator. Syntax:
MUL REGISTER

Notice that only one operant is required. That's because the processor assumes that you want to multiply the give register with AX or AH.

Example:
```
MOV AX,5h
MOV BX,4Fh

MUL BX
```

This leaves AX equal to 18B (4Fh x 5h = 18B). Notice that the result is stored in AX, or AH, depending on what was used for the operation.

Finally we have the DIV operator. Syntax:
```
DIV REGISTER
```

Just like the MUL operator, there is only one operand, and AX is assumed to be the second one.

Example:
```
MOV AX,5h
MOV BX,4Fh

DIV BX
```

Now AX equals Fh since 4Fh / 5h = Fh.
NOTE: The result is rounded to the next lowest number:
```
       4Fh = 79
        5h =  5
       79 / 5 = 15.8

        15 = Fh
```

NOTE: For now it's fine if you use MUL and DIV, but they are very slow operators. That means if you need speed (in graphics for example), NEVER use MUL/DIV! You can use Shifting combined with addition/subtraction to achieve code that can sometimes be 3000% faster! However shifting is a bit difficult to understand if you don't know much about assembly yet, I will completly discuss them in the graphics part of this tutorial.

Bit wise operation

Sounds hard but is very easy. There are 4 bit wise operators: AND, OR, XOR, and NOT. What these do is compare two values bit for bit. This can be extremely useful!

AND syntax:
```
AND REGISTER1, REGISTER2
AND REGISTER, VALUE
```

AND returns 1 (TRUE) only if BOTH operands are 1 (TRUE)

Example 1:
```
MOV AX,5h
MOV BX,6h

AND AX,BX
```

The result is stored in AX. So for this example AX = 4. Lets look at that result more closely:
```
5h = 101b
6h = 110b
```

```
101b
110b
---
100b

100b = 4h

Example 2:
MOV AX,5h

AND AX,6h

The result is the same as in Example 1 (AX = 4h).

AND truth table:
0 AND 0 = 0
1 AND 0 = 0
0 AND 1 = 0
1 AND 1 = 1


OR syntax:
OR REGISTER1, REGISTER2
OR REGISTER, VALUE

OR returns 1 (TRUE) if either operand is 1 (TRUE).

Example 1:
MOV AX,5h
MOV BX,6h

OR AX,BX

AX is now equal to 7h

5h = 101b
6h = 110b

101b
110b
----
111b

111b = 7h

OR truth table:
0 OR 0 = 0
1 OR 0 = 1
0 OR 1 = 1
1 OR 1 = 1


XOR syntax:
XOR REGISTER1, REGISTER2
XOR REGISTER, VALUE

XOR returns 1 (TRUE) if one or the other operand is 1 (TRUE), but not both

Example:
MOV AX,5h
MOV BX,6h

XOR AX,BX
```

AX is not equal to 3h

5h = 101b
6h = 110b

```
101b
110b
----
011b
```

11b = 3h

XOR truth table:
0 XOR 0 = 0
1 XOR 0 = 1
0 XOR 1 = 1
1 XOR 1 = 0

And finally we have NOT. NOT is the easiest one as it simply inverts each bit.

NOT syntax:
NOT REGISTER
NOT VALUE

Example:
MOV AX,F0h

NOT AX

AX is now equal to F since

F0h = 11110000
Invert it:
00001111
which is:
F

NOTE: The windows calculator won't work for this, do it by hand.

NOT truth table:
NOT 1 = 0
NOT 0 = 1

Interrupts

Interrupts are one of the most useful things in assembly. An interrupt is just what it says, a interruption to the normal execution of a program. The best way to illustrate this is one of those "Press any key to continue" things. The program is running but when you press a key it stops for a split second, check what key you pressed and continues. This kind of interrupt is known as a Hardware Interrupt because it uses hardware (the keyboard). The kind of interrupts you will use in your assembly programs are know as Software Interrupts because they are caused by software, not hardware. An example of a software interrupt is reading and writing to a file. This is a DOS interrupt because it is done by DOS, than there are other interrupts done by other things. For example your BIOS or Video Card all have build in interrupts at your exposure. So how does the computer know what interrupt is what? Each interrupt is assigned a number and stored in the Interrupt Vector Table (IVT for short). The IVT is located at 0000:0000 (remember the segment:offset shit. This location would be the origin if plotted on a 2D graph). All interrupt handlers are 1 DWORD in size (double word, 32bit, or 4 bytes). So the handler for interrupt 1h can be found at

0000:0004 (since it's a DWORD it goes up by 4 bytes). The most common interrupt is 21h and can be found at 0000:0084.

So how do you use interrupts?
Very simple:
INT interrupt

For example, in the Program Layout section earlier the program contain the line
INT 20h
The interrupt 20h returns to DOS.

Some interrupts like this one only have one function, but other have many more. So how does the operating system know what function you want? You set the AX register up.

Example:
```
MOV AH,02
MOV DL,41

INT 21
INT 20
```

This program is quite amazing. It prints the character A. Lets make it even better by plugging it into our layout:

```
MAIN SEGMENT
     ASSUME DS:MAIN,ES:MAIN,CS:MAIN,SS:MAIN

START:
    MOV AH,02h
    MOV DL,41h

    INT 21h
    INT 20h
MAIN ENDS
END START
```

Save it and assemble it. Refer back to chapter 2 if you forgot how to do that.
So what is happening here?
First it does the familiar set up, than it set AH to 02, which is the character output function of interrupt 21. Then it moves 41 into DL, 41 is the character A. Finally it calls interrupt 21 which displays the A and quits with interrupt 20.
How do you know what you have to set all those registers to? You get a DOS interrupt list. Check Appendix B for urls.
Quite an accomplishment there, after reading 970 lines of boring text you can finally make a 11 line program that would take 1 line to do in Perl! Pad yourself on that back and lets move on.

Exercises:
1. Make a program that gets the value from AX, puts it into DX and BX, then multiplies the values in DX and BX and stores the result in CX.
   This sounds easier than it really is, use the stack to help you out.
2. Make a program that prints out the string ABC, than quits to DOS
   Hint: A = 41, B = 42, C = 43
3. Make a program that performs ALL bit wise operations using the values 5h and 4Fh

5. Tools

Throughout this tutorial you have been using no software other than your assembler. In this chapter you will learn how to master other software that can be of tremendous help to you.

Debug

Lets start with something that's not only very useful, but also free and already
on your computer.
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 to 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 (B)
-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 (3FD for mine)
Com 2 = 2F8 - 2FF
Com 3 = ??? - ??? (if someone knows, please let me know, I would assume though that it's 0F8 - 0FF.)
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:
AMI/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. Please note however that these are fairly old numbers and BIOS makes constantly change them, so they might not work with your particular BIOS.

Proceed:
Proceeds in the execution of a program, usually used togethter 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 files with a .EXE or .HEX extension.

Enough about debug, lets move on to CodeView.

CodeView

CodeView is another program that might come in handy sometimes. However it is not free. There are many debuggers similar to CodeView out there, but it is enough for you to understand one.
CodeView has a number of different windows, Help, Locals, Watch, Source 1, Source 2, Memory 1, Memory 2, Registers and a few more, depending on the version number.

The Source Windows

Source 1 and 2 let you view 2 different source code segments at the same time. This is very useful for comparing.

Memory Windows

These windows let you view and edit different sections of memory. On the left side you have the memory location in segment:offset form, in the middle the hex value of the instructions, and on the right side the ASCII value. Again, non-printable characters are represented by a ".". You can switch between multiple menus using F6. You can also press Shift+F4 to switch between hexadecimal, ASCII, words, double words, signed integers, floating values, and more.

Register

This menu lets you view and change the value in each register. The FL register near the bottom stands for Flags. At the very bottom you should see 8 different values. They are the specific flag values.

```
OV/NV = Overflow        (OVerflow/No oVerflow)
DN/UP = Direction       (DowN/UP)
DI/EI = Interrupt       (????)
PL/NG = Sign            (????)
NZ/ZR = Zero            (Not Zero/ZeRo)
NA/AC = Auxiliary Carry (No Auxiliary carry/Auxiliary Carry)
PO/PE = Parity          (????)
NC/CY = Cary            (????)
```

Command
This window lets you pass commands to CodeView. I will not explains these as they are almost identical to the ones Debug uses, however a bit more powerful.

This chapter went through a lot of material. Make sure you actually get it all, or at least most of it. Debug will be insanely useful later on, so learn it now! The key is practise, lots of practise!

Exercises:

1. Make a program that prints an A on the screen using debug, save it to C drive as cow.com. Quite debug and delete it. Now get back into debug and restore it again.
   HINT: If you delete a file in DOS, DOS simply changes the first character to E5

   It's not as hard as it sounds, basically here's what you do:
   I) Load as many sectors of your drive as you think you will need
   II) Search those sectors for the hex value E5 and the string "ow"
   III) Dumb the offset of the location the search returned
   IV) Edit that offset and change the E5 instruction to a letter of your choice (41)
   V) Write the sectors you loaded into RAM back to C drive
2. Use debug to get your modem into CS (Clear to Send) mode. The hex value is 2.
3. Make a program called cursor.com using debug that will change the cursor size.
   I) Move 01 into AH
   II) Move 0007 into CX
   III) Call interrupt 10
   IV) Call interrupt 20

6. More basics

Before reading this chapter, make sure you completely understood EVERYTHING I talked about so far.

.COM File Format

COM stands for COre iMage, but it is much easier to memorize it as Copy Of Memory, as that description is even better. A COM file is nothing more than a binary image of what should appear in the RAM. It was originally used for the CP/M and even though CP/M were used in the Z80/8080 period, COM files have still the same features as they did back in the 70's. Let's examine how a COM file is loaded into memory:

1. You type in the file name, DOS searches for filename + .com, if found that file gets executed. If not DOS will search for filename + .exe, if it can't find that it will search for filename + .bat, and if that search fails it will display the familiar "Bad command or filename" message.
2. If it found a .com file in step 1, DOS will check its records and make sure that a 64k block of memory is found. This is necessary or else the new program could overwrite existing memory.

3. Next DOS builds the Program Segment Prefix. The PSP is a 256 byte long block of memory which looks like the table below:

Address	Description
00h-01h	Instructions to terminate the program, usually interrupt 20h
02h-03h	Segment pointer to next available block
04h	Reserved, should be 0
05h-09h	Far call to DOS dispatcher
0Ah-0Dh	INT 22h vector (Terminate program)
0Eh-11h	INT 23h vector (Ctrl+C handler)
12h-15h	INT 24h vector (Critical Error)
16h-17h	PSP segment of parent process
18h-2Bh	Pointer to file handler
2Ch-2Dh	DOS environment segment
2Eh-31h	SS:SP save area
32h-33h	Number of file handles
34h-37h	Pointer to file handle table
40h-41h	DOS version
5Ch-6Bh	File control block 1
6Ch-7Bh	File control block 2
7Ch-7Fh	Reserved
80h	Length of parameter string
81h-FFh	Default DTA (Disk Transfer Area)

4. DS,ES, and SS are set to point to block of memory
5. SP is set to FFFFh
6. 0000h is pushed on the stack (stack is cleared)
7. CS is set to point to memory (segment), IP is set to 0100h (offset, remember debug?)

The PSP is exactly 255 bytes long, meaning that to fit into one segment (aka. to be a valid .com file your program cannot be larger than 65280 bytes). However as I mentioned before, by the time you can code a program in assembly that is that large, you already know well more than enough to make a .EXE file.
So what do you need this information for? Well like all other memory, you can view and edit the PSP. So you could play around with it. For example, later when we get into file operations you will be working with the DTA. Or maybe you need to know the DOS version, you can just check 40h-41h, etc.

Flow control operations

Flow control operations are just what the name says, operations that control the flow of your program. If you have worked with another language before, those are the if/then statements. From what you've hear about assembly, you might think that this is fairly difficult, but it's not. To do the equivalent of a if/then I will have to introduce you to 3 new things, labels, the compare command and jump instructions. First things first, maybe you recall the simple program that prints A from the interrupts section. Notice how our layout contains the line START:? START: is a label. If you come from C/C++/Pascal you can think of a label almost like a function/procedure. Take a look at the following code, by now you should know what's happening here:

```
MAIN  SEGMENT
      ASSUME CS:MAIN,DS:MAIN,SS:NOTHING

      ORG 100h

START:
  INT 20

MAIN ENDS
END START
```

Notice the line that saying START:, that's a label. So what's the point of putting

labels in your code? Simple, you can easily jump to any label in your program using the JMP operator. For example, consider the following code:

```
MAIN  SEGMENT
      ASSUME CS:MAIN,DS:MAIN,ES:MAIN,SS:MAIN

      ORG 100h

START:
  MOV AH,02h
  MOV DL,41h
  INT 21h

  JMP EXIT

  MOV AH,02h
  MOV DL,42h
  INT 21h

EXIT:
  INT 20h

MAIN ENDS
END START
```

First the program prints an A using the familiar routine, but instead of using INT 20h to exit, it jumps to the label EXIT and calls INT 20h from there. The result is that it completely skips everything after the JMP EXIT line, the B doesn't get printed at all. So by using a label you can easily close the program from any location.
This is fairly useless so far though. It gets interesting when you start using some of the other jump commands. I will only explain a few here as there are just too many. Below is a alphabetical list of most of them:

```
JA    - Jump if Above
JAE   - Jump if Above or Equal
JB    - Jump if Below
JBE   - Jump if Below or Equal
JC    - Jump on Carry
JCXZ  - Jump if CX is Zero
JE    - Jump if Equal
JG    - Jump if Greater
JGE   - Jump if Greater than or Equal
JL    - Jump if Less than
JLE   - Jump if Less than or Equal
JMP   - Jump unconditionally
JNA   - Jump if Not Above
JNAE  - Jump if Not Above or Equal
JNB   - Jump if Not Below
JNE   - Jump if Not Equal
JNG   - Jump if Not Greater
JNGE  - Jump if Not Greater or Equal
JNL   - Jump if Not Less
JNLE  - Jump if Not Less or Equal
JNO   - Jump if No Overflow
JNP   - Jump on No Parity
JNS   - Jump on No Sign
JNZ   - Jump if Not Zero
JO    - Jump on Overflow
JP    - Jump on Parity
JPE   - Jump on Parity Even
JPO   - Jump on Parity Odd
JS    - Jump on Sign
```

JZ   - Jump on Zero

Some of these are fairly self-explanatory (like JCXZ), but others require some more explanation. What is being compared to what, and how does the jump know the result? Well almost anything can be compared to almost anything. The result of that comparison is stored in the flags register. The jump command simply checks there and response accordingly. Let's make a simple if/then like structure:

```
MAIN  SEGMENT
      ASSUME CS:MAIN,DS:MAIN,ES:MAIN,SS:MAIN

      ORG 100h

START:
  MOV DL,41h
  MOV DH,41h
  CMP DH,DL
  JE  TheyAreTheSame
  JMP TheyAreNotSame


TheyAreNotSame:
  MOV AH,02h
  MOV DL,4Eh
  INT 21h
  INT 20h

TheyAreTheSame:
  MOV AH,02h
  MOV DL,59h
  INT 21h
  INT 20h
```

This code is fairly straight forward, it could be expressed in C++ as:

```
void main () {
int DH,DL
DL = 41
DH = 41
if (DH == DL) {
    cout << "Y";
 } else {
    cout << "N";
}
```

In this case the program will return Y, but try changing either DH, or DL to some other value. It should display N.

HINT: Tired of constantly typing "tasm blah.asm", "tlink /t blah.obj"? Make a simple batch file containing the following 3 lines and save it as a.bat in your tasm dir.

```
@ECHO OFF
TASM %1.ASM
TLINK /T %1.OBJ
```

Now you can just type "a blah", even without the file extension.
If you have A86 and are sick of typing "a86 blah.asm", just rename a86.exe to something like a.exe.

Loops

Loops are a essential part of programming, in fact loops make the difference between

being a programming language and being something like HTML. If you don't know what loops are, they are just the repeaded execution of a block of code. The 2 most common types of loops are For and While.
A for loop repeads a block of code until a certain condition is met. Look at the following C++ code:

```
main()
{
  for (int counter = 0; counter < 5; counter++)
     {
      cout << "A";
     }
return 0;
```

This will produce the following output:
AAAAA

This code is fairly easy, it initializes a variable and sets it equal to zero. It will loop until the varialbe is less than 5, and after each execution the variable gets incremented. Now lets make an exact copy of that program in assembly:

```
blah            segment
assume          cs:blah, ds:blah, ss:blah, es:blah, ss:blah    ;do the usual setup
org             0100h

start:                                  ;label for start of program
  MOV CX,5                              ;cx is always the counter

LOOP_LABEL:                             ;label to loop
  MOV AH,02h                            ;do the familiar A shit (this printed some wierd
  MOV DL,41h                            ;character instead for me, anyone know why?)
  INT 21h

  LOOP LOOP_LABEL                       ;loop everything in between loop_label: and the
                                        ;loop statement as many times as specified in CX
  INT 20h
                                        ;usual ending shit
blah  ends
end   Start
```

Output should be:
AAAAA
C:\>
But as I said, it printed some other shit for me. Well who cares, as long as it looped.
This code is basicly doing this:
1. Set CX to 5
2. Print an A
3. Check of CX = 0, if not decrement CX
4. Go back to loop_label
5. Check if CX = 0, if not decrement CX
6. etc

Next we have the While loop. It also repeads a block of code as long as a condition is true, but the condition is not changed during in the loop declaration as with the For loop. Take a look at a simple C++ While loop:

```
main()
{
  int counter = 0;
  while(counter < 5)
     {
```

```
      counter++
      cout << "A";
     }
return 0;
```

Notice how the condition is being changed in the actual loop. This is very important as you may already know. Let's convert that piece of code to assembly:

```
blah            segment
assume          cs:blah, ds:blah, ss:blah, es:blah    ;do the usual setup
org             0100h

start:
  MOV CX, 5                                 ;set CX equal to 5
loop_label:
  MOV AH,02h                                ;print the A
  MOV DL,41h
  INT 21h

  DEC CX                                    ;decrement CX

  CMP CX,0                                  ;check if CX is zero
  JNZ loop_label                            ;no?  go back to loop_label

  INT 20h                                   ;yes?  terminate program
                                            ;usual ending shit
blah  ends
end   Start
```

```
Output:
AAAAA
C:\>
```

They look almost identical, so what's different? And why use em? Well the For loop is good for loops that have a set number of repetitions, while the while loop can change the amount or repedition during the loop execution. This is useful for user input for example. It is also possible to make a for loop without using the loop statement, just like you would do a while loop. That might not look as pretty, but it can potentially be a bit faster.

Sometimes interrupts can modify the CX register, in which case your loop would loop an unpredictable number of times. That's not good. To stop that you can make use of the stack:

```
MAIN SEGMENT
     ASSUME CS:MAIN,DS:MAIN,ES:MAIN,SS:MAIN

     ORG 100h

START:
  MOV CX,5

Loop_Label:
  PUSH CX                               ;store CX on the stack
  MOV AH,02h
  MOV DL,41h
  INT 21h
  POP CX                                ;restore it after the pass is completed

  LOOP Loop_Label

  INT 20h
                                        ;usual ending shit
```

```
MAIN ENDS
END START
```

Variables

Yeah, yeah, yeah, I know there are no variables in assembly, but this is close enough for me.
You may be familiar with variables if you've come from another language, if not variables are simply a name given to a memory area that contains data. To access that data you don't have to specify that memory address, you can simply refer to that variable. In this chapter I will introduce you to the 3 most common variables types: bytes, words, and double words. You declare variables in this format:
NAME TYPE CONTENTS
Were type is either DB (Declare Byte), DW (Declare Word), or DD (Declare DoubleWord). Variables can consist of number, characters and underscores (_), but must begin with a character.

Example 1:
```
A_Number DB 1
```

This creates a byte long variable called A_Number and sets it equal to 1

Example 2:
```
A_Letter DB "1"
```

This creates a byte long variable called A_Letter and sets it equal to the ASCII value 1. Note that this is NOT a number.

Example 3:
```
Big_number DD 1234
```

This declares a Double Word long variable and sets it equal to 1234.

You can also create constants. Constants are data types that like variables can be used in your program to access data stored at specific memory locations, but unlike variables they can not be changed during the execution of a program. You declare constants almost exactly like variables, but instead of using D?, you use EQU. Well actually EQU declares a Text Macro. But since we haven't covert macros yet and the effect is basicly the same, we will just tread it as a constant.

Example 4:
```
constant EQU DEADh
```

So how do you use variables and constants? Just as if they were data. Take a look at the next example:

Example 5:
```
constant EQU 100
mov dx,constant
mov ax,constant
add dx,ax
```

This declares a constant called constant and sets it equal to 100, then it assigns the value in constant to dx and ax and adds them. This is the same as
```
mov dx,100
mov ax,100
add dx,ax
```

The EQU directive is a bit special though. It's not really a standard assembly instruction. It's assembler specific. That means that we can for example do the

following:

```
bl0w EQU PUSH
sUcK EQU POP

bl0w CX
sUcK CX
```

When you assemble this, the assembler simply substitues PUSH and POP with every occurance of bl0w and sUcK respectivly.

Arrays

Using this knowledge it is possible to create simple arrays.
Example 1:

```
A_String DB "Cheese$"
```

This creates a 5 byte long array called A_String and sets it equal to the string Cheese. Notice the $ at the end. This has to be there, otherwise your CPU will start executing instructions after the last character, which is whatever is in memory at that particular location. There probably won't be any damage done, but who knows what's hidden in those dark corners...
To use quotes (single or double) within a string you can use a little trick:

Example 2:

```
Cow  DB  'Ralph said "Cheese is good for you!"$'
```

or

```
Cow  DB  "Ralph said 'Cheese is good for you!'$"
```

Use whichever you think looks better. What if you have to use both types of quotes?

Example 3:

```
Cow  DD  'Ralph said "I say: ""GNAAAARF!""$'
```

Use double double/single quotes.

What if you don't know what the variable is going to equal? Maybe it's user-inputed.

Example 4:

```
Uninitialized_variable DB ?
```

Now lets use a variable in an actual program:

```
MAIN  SEGMENT
      ASSUME CS:MAIN,DS:MAIN,ES:MAIN,SS:MAIN

      ORG 100h

START:
COW  DB  "Hello World!$"
  MOV AH,09h
  MOV DX,OFFSET COW
  INT 21h
  INT 20h
MAIN ENDS
END START
```

Yes, even in assembly you finally get to make a Hello World program!
Here we're using interrupt 21h, function 9h to print a string. To use this interrupt you have to set AH to 9h and DX must point to the location of the string.

NOTE: VERY important! ALWAYS declare unitialized arrays at the VERY END of your program, or in a special UDATA segment! That way they will take up no space

at all, regardless of how big you decide to make them. For example say you have this in a program:

```
Some_Data  DB 'Cheese'
Some_Array DB 500 DUP (?)
More_Data  DB 'More Cheese'
```

This will automaticly add 500 bytes of NULL characters to your program. However if you do this instead:

```
Some_Data  DB 'Cheese'
More_Data  DB 'More Cheese'
Some_Array DB 500 DUP (?)
```

Your program will become 500 bytes smaller.

String Operations

Now that you know some basics of strings, let's use that knowledge. There are a number of string operations available to you. Here I will discuss 4 of them. Lets start with MOVSB. This command will move a byte from one location to another. The source destination is ES:SI and the destination is DS:DI.

Example 1:

```
MAIN SEGMENT
     ASSUME CS:MAIN,DS:MAIN,ES:MAIN,SS:MAIN

     ORG 100h

START:
      MOV AH,9

      MOV DX,OFFSET NEWSTRING
      INT 21h

      MOV DX,OFFSET OLDSTRING
      INT 21h

      MOV CX,9
      LEA SI,OLDSTRING
      LEA DI,NEWSTRING

      REP MOVSB

      MOV DX,OFFSET NEWSTRING
      INT 21h

      MOV DX,OFFSET OLDSTRING
      INT 21h

      INT 20h
OLDSTRING DB 'ABCDEFGHI $'
NEWSTRING DB '123456789 $'
MAIN ENDS
END START
```

Output:

```
ABCDEFGHI 123456789 ABCDEFGHI ABCDEFGHI
```

This little example has a few instructions that you haven't seen before, so lets go through this thing step by step.

1. We do the regular setup
2. We use the method from the previous section to print NEWSTRING (ABCDEFGHI)
3. We print OLDSTRING (123456789)
4. We set CX equal to 9. Remeber that the CX register is the counter.

5. Here's a new instruction, LEA. LEA stands for Load Effective Address. This instruction will load the contents of a "variable" into a register. Since DI contains the destination and SI the source, we assign the location of NEWSTRING and OLDSTRING to them respectivly
6. MOVSB is the string operator that will move a byte from SI to DI. Since we have an array of 9 characters (well 10 if you count the space, but that is the same in both anyway) we have to move 9 bytes. To do that we use REP. REP will REPeat the given instruction for as many times as specified in CX. So REP MOVSB will perform the move instruction 9 times, ones for each character.
7. To see our result we simple print each string again using the same code we used in step 2 and 3.

The next string operator is not only very easy to use, but also very useful. It will scan a string for a certain character and set the EQUAL flag bit if the search was successful. The operator is SCASB, the location of the string is in DI, and the character is stored in AL.

Example 2:

```
MAIN SEGMENT
     ASSUME CS:MAIN,DS:MAIN,ES:MAIN,SS:MAIN

     ORG 100h

START:
     MOV CX,17h
     LEA DI, STRING
     MOV AL, SEARCH

     REPNE SCASB

     JE  FOUND
     JNE NOTFOUND


NOTFOUND:
     MOV AH,09h
     MOV DX,OFFSET NOTFOUND_S
     INT 21h
     INT 20h

FOUND:
     MOV AH,09h
     MOV DX,OFFSET FOUND_S
     INT 21h
     INT 20h

SEARCH     DB '!'
STRING     DB  'Cheese is good for you!'
FOUND_S    DB  'Found$'
NOTFOUND_S DB  'Not Found$'

MAIN ENDS
END START
```

This should be fairly easy to figure out for you. If you can't, I'll explain it:
1. We do the usual setup
2. We set CX equal to 17h (23 in decimal), since our string is 17h characters long
3. We load the location of STRING into DI
4. And the value of the constant SEARCH into AL
5. Now we repeat the SCASB operation 23x
6. And use a jump to signal wether or not we found the string

Finally we have the CMPS instruction. This operator will compare the value of two strings with each other until they're equal.

Example 3:

```
MAIN SEGMENT
     ASSUME CS:MAIN,DS:MAIN,ES:MAIN,SS:MAIN

     ORG 100h

START:
     MOV CX,17h
     LEA SI,STRING1
     LEA DI,STRING

     REP CMPSB

     JE  EQUAL
     JNE NOTEQUAL

NOTEQUAL:
     MOV AH,09h
     MOV DX,OFFSET NOT_EQUAL
     INT 21h
     INT 20h

EQUAL:
     MOV AH,09h
     MOV DX,OFFSET EQUAL1
     INT 21h
     INT 20h

STRING1    DB 'Cheese is good for you!'
STRING     DB 'Cheese is good for you!'
EQUAL1     DB 'They''re equal$'
NOT_EQUAL  DB 'They''re not equal$'

MAIN ENDS
END START
```

By now you should know what's going on. SI and DI contain the two strings to be compared, and REP CMPSB does the comparison 17h times, or until it comes across two bytes that are not equal (b and g in this case). Then it does a jump command to display the appropriate message.

The final string operations I will introduce you to are STOSB and and LODSB. STOSB will store a byte from AL at the location that ES:DI points to. STOSB will get a byte that ES:DI points into AL. These two instructions are very very powerful as you will see if you continue learning assembly. Take a look at the next example.

```
MAIN SEGMENT
     ASSUME CS:MAIN,DS:MAIN,ES:MAIN,SS:MAIN

     ORG 100h

START:
     MOV AH,9
     MOV DI,OFFSET STRING

     MOV SI,DI

     LODSB
     INC AL
     STOSB
```

```
     MOV DX,DI
     DEC DX

     INT 21h
     INT 20h

STRING DB "oh33r! $"

MAIN ENDS
END START
```

This code will return:
ph33r!

So what does it do?
1. It moves 9 into AH to set it up for interrupt 21's Print String function
2. Move the location of STRING into DI for the the LODSB instruction
3. Do the same with SI
4. Load ES:DI into AL
5. Increment AL, thus changing it from o to p
6. And put the contents of AL back to ES:DI
7. Put DI into DX for interrupt 21's Print String function
8. STOSB will increment DI after a successful operation, so decrement it
9. Call interrupt 21h
10. And terminate the program

And here's a final note that I should have mentioned earlier: All these string instructions actually don't always end in B. The B simply means Byte but could be replaced by a W for example. That is, MOVSB will move a byte, and MOVSW will move a word. If you're using a instruction that requires another register like AL for example, you use that registers 32 or 64 bit part. For example, LODSW will move a word into AX.

Sub-Procedures

This chapter should be fairly easy as I will only introduce one new operator, CALL. CALL does just that, it CALLs a sub-procedure. Sub-Procedure are almost exactly like labels, but they don't end with a : and have to have a RET statement at the end of the code. The purpose of sub-procedures is to make your life easier. Since you can call them from anywhere in the program you don't need to write certain sections over and over again.

```
MAIN SEGMENT
     ASSUME CS:MAIN,DS:MAIN,ES:MAIN,SS:MAIN

     ORG 100h

START:
       CALL CHEESE
       CALL CHEESE
       CALL CHEESE
       CALL CHEESE
       CALL CHEESE
       CALL CHEESE
       CALL CHEESE
       INT 20h

CHEESE PROC
       MOV AH,09
       LEA DX,MSG
       INT 21h
```

```
       RET
CHEESE ENDP

MSG DB 'Cheese is good for you! $'

MAIN ENDS
END START
```

1. We use the CALL command to call the sub-procedure CHEESE 7 times.
2. We set up a sub-procedure called CHEESE. This is done in the following format:
   LABEL PROC
3. We type in the code that we want the sub-procedure to do
4. And add a RET statement to the end. This is necessary as it returns control to the main function. Without it the procedure wouldn't end and INT 20h would never get executed.
5. We end the procedure using
   LABEL ENDP
6. The usual...

User Input

Finally! User Input has arrived. This chapter will discuss simple user input using BIOS interrupts. The main keyboard interrupt handler is 16h. For the first part of this chapter we will be using the function 0h.
Lets start with a simple program that waits for a keypress:

Example 1:

```
MAIN SEGMENT
     ASSUME CS:MAIN,DS:MAIN,ES:MAIN,SS:MAIN

     ORG 100h

START:
    MOV AH,0
    INT 16h
    INT 20h
MAIN ENDS
END START
```

This program waits for you to press a key, and then just quits. Expected more? Of course. We have the echo the key back. Only than it will be truly 3|337. Remember all those programs you did in the debug part of this tutorial that printed out an A? Remember how we did it? No? Like this:

Example 2:

```
MAIN SEGMENT
     ASSUME CS:MAIN,DS:MAIN,ES:MAIN,SS:MAIN

     ORG 100h

START:
    MOV AH,2h
    MOV DL,41h
    INT 21h
    INT 20h
MAIN ENDS
END START
```

Notice how the register DL contains the value that we want to print. Well if we use interrupt 16h to get a key using function 0h, the ASCII scan code gets stored in AL, so all we have to do is move AL into DL, then call the old interrupt 21h, function 2h.

Example 3:

```
MAIN SEGMENT
     ASSUME CS:MAIN,DS:MAIN,ES:MAIN,SS:MAIN

     ORG 100h

START:
    MOV AH,0h
    INT 16h

    MOV AH,2h
    MOV DL,AL
    INT 21h
    INT 20h
MAIN ENDS
END START
```

Isn't this awesome? Well that's not all INT 16 can do. It can also check the status of the different keys like Ctrl, Alt, Caps Lock, etc. Check Appendix A for links to interrupt listings and look them up.
Let's use our new found t3kn33kz to create another truly 3|337 program:

Example 4:

```
MAIN SEGMENT
     ASSUME CS:MAIN,DS:MAIN,ES:MAIN,SS:MAIN

     ORG 100h

START:
    MOV AH,0h
    INT 16h
    MOV KEY,AL

    CMP KEY,90h
    JE ITS_A_Z
    JNE NOT_A_Z

ITS_A_Z:
    MOV AH,9h
    MOV DX,OFFSET NOTA
    INT 21h

    INT 20h

NOT_A_Z:
    MOV AH,2h
    MOV DL,KEY
    INT 21h

    INT 20h

KEY DB ?
NOTA DB "You pressed Z!!!!!!!!",10,13,"Ph33r! $"
MAIN ENDS
END START
```

Well you should be able to understand this program without any problems. If you don't:
1. We set AX to 0h and call interrupt 16h, that waits for the user to press a key
2. We move the value of AL (which holds the ASCII value of the key pressed) into a variable. This way we can manipulate the registers without having to worry about destroying it
3. We compare KEY with 90h, which is hex for Z (case sensitive)
4. If it is a Z we jump to ITS_A_Z which displays the message

5. If not, we jump to NOT_A_Z, which simply echos the key back.
6. We decalared 2 variables, one which is not initialized yet called KEY, and one
    that holds the value "You pressed Z!!!!!!!!",10,13,"Ph33r! $" Which looks like
    this on a DOS computer:
    You pressed Z!!!!!!!!
    Ph33r!

Exercises:
1. Make a program that will accept a series of keypresses, but when the user enters
    the following characters, convert them to their real values as shown below:
    S = Z
    F = PH
    PH = F
    E = 3
    I = 1
    EA = 33
    T = 7
    O = 0
    A = 4
    L = |
    NOTE: This is NOT case sensitive. In other words, you're going to either have to
    convert lower case to upercase (or the otherway around) as soon as its entered by
    for example subtracting 20 from the ASCII value, or by making a branch for either
    case.
    Also, try using procedures to do this.

7. Basics of Graphics

Graphics are something we all love. Today you will learn how to create some bad ass graphics in assembly! Well actually I will tell you how to plot a pixel using various methods. You can apply that knowledge to create some other graphics routines, like line drawing shit, or a circle maybe. It's all just grade 11 math.

Using interrupts

This is the easiest method. We set up some registers and call an interrupt. The interrupt we will be using is 10h, BIOS video. Before we do anything, we have to get into graphics mode. For the purpose of simplicity I will just cover 320x200x256 resolution (that is 320 vertical pixels, 200 horizontal pixels, and 256 shades of colors). So how do you get into this mode? You set AH to 00h and AL to 13h. 00h tells interrupt 10h that we want to get into graphics mode, and 13h is the mode (320x200x256).

Example 1:

```
MAIN SEGMENT
     ASSUME DS:MAIN,ES:MAIN,SS:MAIN,CS:MAIN

     ORG 100h

START:
    MOV AH,00h
    MOV AL,13h
    INT 10h

    INT 20h
MAIN ENDS
END START
```

This ins't too exiting, just looks bigger. Let's plot a pixel.

Example 2:

```
MAIN SEGMENT
     ASSUME DS:MAIN,ES:MAIN,SS:MAIN,CS:MAIN

     ORG 100h

START:
    MOV AH,00h
    MOV AL,13h
    INT 10h

    MOV AH,0Ch
    MOV AL,10
    MOV CX,100
    MOV DX,100
    MOV BX,1h
    INT 10h

    INT 20h
MAIN ENDS
END START
```

First we get into graphics mode, then we set AH to 0Ch which is the Draw Pixel function of interrupt 10h. In order to use 0Ch we have to set up some other registers as well. AL contains the colors of the pixel, CX the location on the X axis and DX the location on the Y axis. Finally BX tells interrupt 10h to use page one of the VGA card. Don't worry about what pages are until you get into more advanced shit.
Once in graphics mode you can switch back to text using

```
    MOV AH,00h
    MOV AL,03h
    INT 10h
```

So putting it all together, the following program will draw a green pixel at location 100,100 on page 1, then switch back to text mode, clearing the pixel along the way. Notice that it sets the AL and AH registers using only 1 move by moving them into AX. This might save you a clock tick or two and makes the executable file a whooping 3 bytes smaller!

Example 3:

```
MAIN SEGMENT
     ASSUME DS:MAIN,ES:MAIN,SS:MAIN,CS:MAIN

     ORG 100h

START:
    MOV AX,0013h
    INT 10h

    MOV AX,0C04h
    MOV CX,100
    MOV DX,100
    MOV BX,1h
    INT 10h

    MOV AX,0003h
    INT 10h

    INT 20h
MAIN ENDS
END START
```

Even though we did a bit of optimization there, it's still very slow. Maybe with one

pixel you won't notice a difference, but if you start drawing screen full after screen full using this method, even a fast computer will start to drag. So lets move on to something quite a bit faster.
By the way, if you're computer is faster than a 8086, you will see nothing at all because even though the routine is slow, a single pixel can still be drawn fast. So the program will draw the pixel and earase is before your eye can comprehend its existance.

Writing directly to the VRAM

This is quite a bit harder than using interrupts as it involves some math. To make things even worse I will introduce you to some new operators that will make the pixels appear even faster.
When you used interrupts to plot a pixel you were just giving the X,Y coordinates, when writing directly the the VRAM you can't do that. Instead you have to find the offset of the X,Y location. To do this you use the following equation:
Offset = Y x 320 + X
The segment is A000, which is were VRAM starts, so we get:
A000:Y x 320 + X
However computers hate multiplication as it is just repeated adding, which is slow. Let's break that equation down into different numers:
A000:Y x 256 + Y x 64 + X
or
A000:Y x 2^8 + Y x 2^6 + X
Notice how now we're working with base 2? But how to we get the power of stuff? Using Shifts. Shifting is a fairly simple concept. There are two kinds of shifts, shift left and shift right. When you shift a number, the CPU simply adds a zero to one end, depending on the shift that you used. For example, say you want to shift 256

```
       256 = 100000000b
Shift Left:  1000000000b
       512 = 1000000000b
Shift Right: 0100000000b
       256 = 100000000b
```

Shifts are equal to 2^n where N is the number shifted by. So we can easily plug shifts into the previous equation.
A000:Y SHL 8 + Y SHL 6 + X
This is still analog. Let's code that in assembly:

```
SET_VSEGMENT:                       ;set up video segment
    MOV AX,0A000h                   ;point ES to VGA segment
    MOV ES,AX

VALUES:                             ;various values used for plotting later on
    MOV AX,100                      ;X location
    MOV BX,100                      ;Y location

GET_OFFSET:                         ;get offset of pixel location using X,Y
    MOV DI,AX                       ;put X location into DI
    MOV DX,BX                       ;and Y into DX
    SHL BX,8                        ;Y * 2^8.  same as saying Y * 256
    SHL DX,6                        ;Y * 2^8.  same as sayinh Y * 64
    ADD DX,BX                       ;add the two together
    ADD DI,BX                       ;and add the X location
```

Now all we have to do is plot the pixel using the STOSB instruction. The color of the pixel will be in AL.

```
    MOV AL,4                        ;set color attributes
    STOSB                           ;and store a byte
```

So the whole code to plot a pixel by writing directly to the VRAM looks like this:

```
MAIN SEGMENT
     ASSUME CS:MAIN,ES:MAIN,DS:MAIN,SS:MAIN

     ORG 100h

START:
    MOV AH,00h                      ;get into video mode.  00 = Set Video Mode
    MOV AL,13h                      ;13h = 320x240x16
    INT 10h

SET_VSEGMENT:                       ;set up video segment
    MOV AX,0A000h                   ;point ES to VGA segment
    MOV ES,AX

VALUES:                             ;various values used for plotting later on
    MOV AX,100                      ;X location
    MOV BX,100                      ;Y location

GET_OFFSET:                         ;get offset of pixel location using X,Y
    MOV DI,AX                       ;put X location into DI
    MOV DX,BX                       ;and Y into DX
    SHL BX,8                        ;Y * 2^8.  same as saying Y * 256
    SHL DX,6                        ;Y * 2^8.  same as sayinh Y * 64
    ADD DX,BX                       ;add the two together
    ADD DI,BX                       ;and add the X location
;this whole thing gives us the offset location of the pixel

    MOV AL,4                        ;set color attributes
    STOSB                           ;and store

    XOR AX,AX                       ;wait for keypress
    INT 16h

    MOV AX,0003h                    ;switch to text mode
    INT 10h

    INT 20h                         ;and exit

END START
MAIN ENDS
```

If you don't understand this yet, study the source code. Remove all comments and add them yourself in your own words. Know what each line does and why it does what it does.

A line drawing program

To finish up the graphics section I'm going to show you a little modification to the previous program to make it print a line instead of just a pixel. All you have to do is repeat the pixel ploting procedure as many times as required. It should be commented well enough, so I wont bother explaining it.

```
MAIN SEGMENT
     ASSUME CS:MAIN,ES:MAIN,DS:MAIN,SS:MAIN

     ORG 100h

START:
    MOV AH,00h                      ;get into video mode.  00 = Set Video Mode
    MOV AL,13h                      ;13h = 320x240x16
    INT 10h

SET_VSEGMENT:                       ;set up video segment
```

```
    MOV AX,0A000h                   ;point ES to VGA segment
    MOV ES,AX

VALUES:                             ;various values used for plotting later on
    MOV AX,100                      ;X location
    MOV BX,100                      ;Y location
    MOV CX,120                      ;length of line.  used for REP

GET_OFFSET:                         ;get offset of pixel location using X,Y
    MOV DI,AX                       ;put X location into DI
    MOV DX,BX                       ;and Y into DX
    SHL BX,8                        ;Y * 2^8.  same as saying Y * 256
    SHL DX,6                        ;Y * 2^8.  same as sayinh Y * 64
    ADD DX,BX                       ;add the two together
    ADD DI,BX                       ;and add the X location
;this whole thing gives us the offset location of the pixel

    MOV AL,4                        ;set color attributes
    REP STOSB                       ;and store 100 bytes, decrementing CX and
                                    ;incrementing DI

    XOR AX,AX                       ;wait for keypress
    INT 16h

    MOV AX,0003h                    ;switch to text mode
    INT 10h

    INT 20h                         ;and exit

END START
MAIN ENDS
```

8. Basics of File Operations

In the old days, DOS did not include interrupts that would handle file operations. So programers had to use some complicated t3kn33kz to write/open files. Today we don't have to do that anymore. DOS includes quite a few interrupts to simplify this process.

File Handles

File handles are are numbers assigned to a file upon opening it. Note that opening a file does not mean displaying it or reading it. Take a look at the following code:

```
MAIN SEGMENT
     ASSUME CS:MAIN,DS:MAIN,SS:MAIN,ES:MAIN
     ORG 100h

START:
     MOV AX,3D00h
     LEA DX,FILENAME
     INT 21h

     JC ERROR

     INT 20h

ERROR:
     MOV AH,09h
     LEA DX,ERRORMSG
     INT 21h
```

```
    INT 20h

FILENAME  DB 'TEST.TXT',0
ERRORMSG  DB 'Unable to open [test.txt]$'
MAIN ENDS
END START
```

If you have a file called test.txt in the current directory the program will simply quite. If the file is missing it will display an error message. So what's happening here?

1. We move 3D00h into AX. This is a shorter way of saying:
   MOV AH,3Dh
   MOV AL,00h
   3Dh is the interrupt 21h function for opening files. The interrupt checks the AL register to how it should open the file. The value of AL is broken down into the following:

   Bit 0-2: Access mode
   0 - Read
   1 - Write
   2 - Read/Write

   Bit 3: Reserved (0)

   Bit 4-6: Sharing Mode
   0 - Compadibility
   1 - Exclusiv
   2 - Deny Write
   3 - Deny Read
   4 - Deny None

   Bit 7: Inheritance Flag
   0 - File is inherited by child processes
   1 - Prive to current process

   Don't worry too much about what all this means. We will only use the Access mode bit.
2. We load the address of the file name into DX. Note that the filename has to be an ASCIIZ string, meaning it is terminated with a NULL character (0).
3. We call interrupt 21h
4. If an error occured while opening the file, the carry flag is set and the error code is returned in AX. In this case we jump to the ERROR label.
5. If no error occured, the file handel is stored in AX. Since we don't know what to do with that yet, we terminate the program at this point.

Reading files

Having optained the file handle of the file, we can now use the file. For example read it. When you use interrupt 21h's file read function, you have to set up the registers as follows:

AH = 3Fh
BX = File handle
CX = Number of bytes to read
DX = Pointer to buffer to put file contents in

Than you simply print out that buffer using interrupt 21h's print string function. However notice how you have to specify the amount of data to read. That's not good since most of the time we don't know how much data is in a file. So we can use a little trick. If an error occured, the error code is stored in AX. The error code 0 means that the program has encounter a EOF (End Of File). So we can simply make a

while loop that prints a single byte from the text as long as AX is not equal to zero. If it is, we know that the file ended and we can terminate the program. Note that it is good coding practise to use interrupt 21h's function Close File to do just that. Here is the code for this thing:

```
MAIN SEGMENT
     ASSUME CS:MAIN,DS:MAIN,SS:MAIN,ES:MAIN
     ORG 100h

START:
     MOV AX,3D00h
     LEA DX,FILENAME
     INT 21h

     JC ERROR

     MOV BX,AX

READFILE:
     MOV AH,3Fh
     MOV CX,0001h
     LEA DX,CHARACTER
     INT 21h

     CMP AX,000h
     JE ENDPROGRAM

     MOV AH,02h
     MOV DL,CHARACTER
     INT 21h

     JMP READFILE

ENDPROGRAM:
     MOV AH,3Eh
     INT 21h

     INT 20h

ERROR:
     MOV AH,09h
     LEA DX,ERRORMSG
     INT 21h

     INT 20h

FILENAME  DB 'TEST.TXT',0
ERRORMSG  DB 'Unable to open [test.txt]$'
CHARACTER DB ?
MAIN ENDS
END START
```

This is a fairly big piece of code, but you should be able to understand it.

1. We get the file handle using the method discussed in the previous chapter.
2. We move the file handle from AX into BX. This is because interrupt 21h's function to read a file requires the handle to be in BX.
3. We move 3Fh into AH, tells interrupt 21h that we want to read a file
4. CX contains the bytes to read, we only want one
5. The read byte is put into buffer that DX points to. In this case its called CHARACTER. Notice how we set up CHARACTER is an unitialized variable.
6. We compare AX to 0, which it would be if a EOF is encountered. If it is, we end the program.

7. Otherwise we use interrupt 21h's function Print Character to print the character
   in the buffer. You should be familiar with that from previous chapters.
8. We return to the label READFILE to read another byte.
9. If EOF is encountered, we use function 3Eh to close the file and terminate the
   program.

Creating files

To create files you have to:
1. Create an empty file
2. Move a buffer into the file handle
The following code will do that for us:

```
MAIN SEGMENT
     ASSUME CS:MAIN,DS:MAIN,ES:MAIN,SS:MAIN

     ORG 100h

START:
     MOV AH,3Ch
     XOR CX,CX
     MOV DX,OFFSET FILE_NAME
     INT 21h

     JC ERROR1

     MOV BX,AX
     MOV AH,40h
     MOV CX,9
     MOV DX,OFFSET SHIT
     INT 21h

     JC ERROR

     INT 20h

ERROR:
     MOV AH,09h
     MOV DX,OFFSET ERROR_WRITING
     INT 21h
     INT 20h

ERROR1:
     MOV AH,09h
     MOV DX,OFFSET ERROR_CREATING
     INT 21h
     INT 20h

FILE_NAME       db "blow.me",0
SHIT            db "123456789"
ERROR_WRITING   db "error writing to file$"
ERROR_CREATING  db "error creating file$"
MAIN ENDS
END START
```

1. We create the file using function 3Ch. Register have to be set up like this:
   CX - Type of file. 0 - normal, 1 - Read Only, 2 - Hidden
   DX - Name of the file. Has to be an ASCIIZ string.
   This function returns the file handle of the new file in AX.
2. We check for an error, and jump of necessary

3. We move the file handle from AX into BX
4. And choose interrupt 21h's function Write File (40h). For this function we need the registers set up like this:
   BX - File Handle
   CX - File size to write (9 in our case)
   DX - Points to buffer to be written
5. We check for an error, if so we jump, otherwise we terminate the program.

Search operations

In assembly you have two search functions at your disposal, Search First and Search Next. Out of those to search first is the more complicated one. As the name implies, Search Next can only be done after a Search First function. So first thing we do to search for a file is set up a Search First routine. The register have to be setup as follows:
AH - 4Eh
CL - File Attributes
DX - Pointer to ASCIIZ path/file name
The file attributes are set up in a wierd way, and I will not get into those. It's enough for you to know that we will be using 6h, which is a normal file. Well actually DOS will read 6h as 00000110, and each bit has a different meaning.
This function will return an error in AX. If AX is zero the search was successful, otherwise we know it wasn't. If it found the files to search for, DOS will setup a block of memory 43 bytes long in the DTA. DTA stands for Disk Transfer Area and for now it's enough to think of it as a "scratch pad" for DOS. In this tutorial I will not get into reading it, but it doesn't hurt telling you what these 43 byte contain:
0 - 21: Reserved for the Find Next function. This saves us from having to do the setup again.
21 - 22: Attributes of the file found
22 - 24: Time the file found was created
24 - 26: Date the file found was created
26 - 30: Size of the file found (in bytes)
30 - 43: File name of the file found.

So our Search First function will look like this:
```
SEARCH:
    MOV DX,OFFSET FILE_NAME
    MOV CL,6h
    MOV AH,4Eh
    INT 21h
    OR  AL,AL
    JNZ NOT_FOUND
```
Notice how we use a bitwise operator instead of a CMP? Bitwise operations are insanly fast, and CoMParing 2 values is bascily subtracting which is slower. Remember how OR works?
0 OR 0 = 0
1 OR 0 = 1
0 OR 1 = 1
1 OR 1 = 1
So OR AL,AL will only return 0 if every single bit in AL is 0. So if it doesn't return 0, we know that it contains an error code and the search failed. We wont bother checking what the error code is, we just jump to a label that will display an error message. If the search was successful we move on to the Search Next function to check if anymore files meet our describtion. Search Next is a fairly easy function. All we have to do is move 4Fh into AH and call int 21h.
```
FOUND:
    MOV Ah,4Fh
    INT 21h
    OR AL,AL
    JNZ ONE_FILE
```

This code will perform the Search Next function, and if it fails jump to the label ONE_FILE. But what happens if it found another file? Well we could do another Search Next function.

```
MORE_FOUND:
    MOV AH,4Fh
    INT 21h
    OR AL,AL
    JNZ MORE_FILES_MSG
```

This will check if yet another file is found. Now we should implement a way of knowing how many files we found. We can do so by setting a register to 1 after the Search First function was successful, and incremeant it each time it finds another file. So lets put all this together and create a program that will search for a file, search again if it found it and start a loop that keeps searching for files and keeps track of how many it found:

```
MAIN SEGMENT
     ASSUME CS:MAIN,DS:MAIN,ES:MAIN,SS:MAIN

     ORG 100h

SEARCH:
    MOV DX,OFFSET FILE_NAME
    MOV CL,6h
    MOV AH,4Eh
    INT 21h
    OR AL,AL
    JNZ NOT_FOUND

FOUND:
    MOV CL,1                ;the counter that keeps track of how many files we found
    MOV Ah,4Fh
    INT 21h
    OR AL,AL
    JNZ ONE_FILE

MORE_FOUND:
    INC CL                  ;here we increment it
    MOV AH,4Fh
    INT 21h
    OR AL,AL
    JNZ MORE_FILES_MSG
    JMP MORE_FOUND

MORE_FILES_MSG:
    MOV AH,02h
    OR  CL,30h              ;convert counter to number (see blow)
    MOV DL,CL               ;and display it.
    INT 21h

    MOV AH,9h
    MOV DX,OFFSET MORE_FILES
    INT 21h
    INT 20h

ONE_FILE:
    MOV AH,9h
    MOV DX,OFFSET FILE_FOUND
    INT 21h
    INT 20h

NOT_FOUND:
    MOV AH,9h
```

```
    MOV DX,OFFSET FILE_NOT_FOUND
    INT 21h
    INT 20h


MORE_FILES     DB " FILES FOUND",10,13,'$'
FILE_NOT_FOUND DB "FILE NOT FOUND",10,13,'$'
FILE_FOUND     DB "1 FILE FOUND",10,13,'$'
FILE_NAME      DB "*.AWC",0                    ;this is the file we search for

MAIN ENDS
END SEARCH
```

Returns:
FILE NOT FOUND
If not files with extension .AWC are found
1 FILE FOUND
If the current directory contains 1 file with the extension .AWC
X FILES FOUND
If more than one file with extension .AWC was found. X stands for the number of files found. Remember how function 2h will print the ASCII value of a hex number? Well we don't really want that. So to convert it to a number we OR it with 30h. That's because if you look at an ASCII chart you'll notice that the numeric value of a ASCII number is always 30h more than the hex number. For example, The number 5 is equal to 35h, 6 is 36h, etc. So to convert it we OR it with 30h:

```
 5h = 000101
30h = 000110
35h = 110101     (ASCII Value: "5")

 6h = 000110
30h = 110000
36h = 110110     (ASCII Value: "6")
```

etc.

Exercises:
1. Create a program that will display how many files are in the current directory
2. Create a program that will create a new file, write something to it, close it, open it, and read its contents.

Basics of Win32

Introduction

I didn't want to include this as I absolutly HATE microsoft, but I guess I have to face the fact that it sadly took over all other good operating systems and people have started to switch to it. This chapter will be quite a bit different from the previous ones as Win32 programing is not really low level. Basicly all you're doing is making calls to internal windows .DLL files. But the most signicant differance is the fact that you will be working in Protected Mode. This is the mode a briefly mentioned where you have a 4 gig limit instead of the old 64k you've been working with so far. I won't heavily get into what protected mode is and does as that is out of the scope of this tutorial (my next asm tutorial will though), but you will need to refer back to .EXE file layout I talked about in chapter 3.

Tools

Well first of all you will have to download a new assembler. That's because my version of TASM is older and doesn't support Win32. So for this chapter get yourself a copy of MASM. That's an assembler by microsoft that has now become freeware. Why didn't I mention MASM before since it's free? Well the only thing MASM is now good for is Win32 programing. TASM uses something called IDEAL mode which is a much better way of programing in assembly. MASM uses MASM mode which quite frankly blows. Get MASM from: <url>
Download and install it, than move on to the next section

A Message Box

First of all you have to get familiar with the program layout:

```
386
.MODEL Flat, STDCALL

.DATA

.DATA?

.CONST

.CODE

LABEL:

END LABEL
```

This should look fairly familiar to you. If it doesn't, let's go over it again:

386 - This declares the processor type to use. You can also use 4 and 586, but for the sake of backwards compadibility you should stick with 386 unless you have to use something higher.

.MODEL FLAT, STDCALL - This declares the memory model to use. In Win32 program you don't have the choices you did before anymore, FLAT is the only one. The STDCALL tells the assembler how to pass parameters. Don't worry about what that means just yet, you will most likely never use anything buy STDCALL in Win32 programming as there is only 1 instruction that needs a different one (C).

.DATA - All your initialized data should go in here

.DATA? - All your uninitialized data should go here

.CONSTS - Constants go here

.CODE - And your code goes here

LABEL: - Just like before, you have to define a starting label

END LABEL - And END it

Now I'm gonna ask you to take a different look at this whole assembly thing. So far you have been manipulating memory and the CPU, with Win32 you manipulate memory and Windows components. I'm sure you know what Include files are, files that will be included with your program when you compile it. Well in Win32 programing you're using windows include files in the form of DLLs. These files are known as Application Programming Interface or API for short. For example, Kernel32.dll, User32.dll, gdi32.dll are APIs. Again, I won't bother getting into details on how APIs work. Assuming you have included all

the .DLL files you need, you call specific Win32 functions in the following format:

```
INVOKE expression,arguments
```

So for example, to exit a program by making a call to the exit function you do:

```
INVOKE ExitProcess,0
```

So let's make a program that does just that, exits:

```
386
.MODEL Flat, STDCALL
option casemap:none                          ;turn case sensitivity on
include \masm32\include\windows.inc          ;the include files that we need
include \masm32\include\kernel32.inc
includelib \masm32\lib\kernel32.lib
.DATA
.CODE

START:
INVOKE ExitProcess,0
END START
```

To get an .EXE out of this, get into your MASM directory and then into BIN. Then assemble with:

```
ml  /c  /coff  /Cp filename.asm
```

And link with:

```
link /SUBSYSTEM:WINDOWS  /LIBPATH:c:\masm32\lib  filename.obj
```

This will get you a file called filename.exe, run it and ph33r.
Now lets make this into a message box. We use the INVOKE command again, but instead of using the ExitProcess function, we use MessageBox.

```
INVOKE MessageBox, 0, OFFSET MsgBoxText, OFFSET MsgBoxCaption, MB_OK
```

Let's disect this thing:
MessageBox tells windows what function we want, and add a 0, just like we did with ExitProcess. This is done because all ANSI strings in windows must be terminated with a 0. Next we put the location of MsgBoxText in there. This is done just like you would do it using INT 21h, OFFSET LOCATION. We do the same with MsgBoxCaption and finally specify what kind of message box we want. In this case MB_OK is a constant representing the familiar box where you can only press Ok. Usually this would be a number, but we're including a file that contains defintions of them. So how did I know what goes where? A Win32 refrence will tell you. We also have to define MsgBoxText and MsgBoxCaption. We do this the way we always did:

```
MsgBoxCaption DB "ph33r b1ll g473z!",0
MsgBoxText    DB "Yes, I ph33r",0
```

So throwing it all together, the code would look like this:

```
.386
.MODEL FLAT,stdcall
option casemap:none
include \masm32\include\windows.inc
include \masm32\include\kernel32.inc
includelib \masm32\lib\kernel32.lib
include \masm32\include\user32.inc
includelib \masm32\lib\user32.lib

.DATA
MsgBoxCaption DB "ph33r b1ll g473z!",0
MsgBoxText    DB "Yes, eYe ph33r",0
```

```
.CODE
START:
INVOKE MessageBox, 0, OFFSET MsgBoxText, OFFSET MsgBoxCaption, MB_OK
INVOKE ExitProcess, 0
END START
```

NOTE: Instead of offset you could have use ADDR. ADDR does basicly the same, but it can handle forward refrences and OFFSET can't. In other words, if you would have declared MsgBoxCaption and MsgBoxText after you use them (INVOKE.....), using OFFSET would return an error. So you should get the habbit of using ADDR instead of Win32.

Now assemble and link with:

```
ml  /c  /coff  /Cp filename.asm
link /SUBSYSTEM:WINDOWS  /LIBPATH:c:\masm32\lib  filename.obj
```

By the way, you should have made a .bat file by now that does this for you. If you haven't, make a file containing the following lines and save it as whatever.bat:

```
@echo off
ml  /c  /coff  /Cp %1.asm
link /SUBSYSTEM:WINDOWS  /LIBPATH:c:\masm32\lib  %1.obj
```

A Window

As I said, I hate Win32 programming, so I'm thinking of scratching this part as it's quite a bit more complex than a message box. E-mail me with your opinion, if enough people think I should do it, I will. So far I have recieved no E-mail of any kind.

Appendix A:
Resources

http://awc.rejects.net
  My homepage, check the sk00lingz and k0d3 sections. Also, this is where you will find the newest version of this and other tutorials by myself. So check it out as some of the other sites that have this tutorial might not be updating it regularly.

http://www.intel.com
  Good selection of white papers on intel's CPUs

http://www.borland.com
  Homepage of the makers of TASM

http://asmjournal.freeservers.com/
  Assembly E-Zine. Very good. Also has a few links to other assembly sites, which than link to even more, which link to still more....

http://www.sandpile.org/
  Good info on hardware programming

http://webster.cs.ucr.edu/Page_asm/ArtofAssembly/ArtofAsm.html
  Very good book on assembly, although it's MASM specific, most t3kn33kz apply to TASM as well.

http://grail.cba.csuohio.edu/~somos/asmx86.html
  Assembly links

http://www.cs.cmu.edu/afs/cs.cmu.edu/user/ralf/pub/WWW/
  Ralph Brown's website. He made a huge listing of interrupts, go there now!

http://www.packetstorm.securify.com/papers.html
  Has a few links to ASM related shit, mostly other shit though. Kick ass site.

http://www.crackstore.com
  Has a shitload of tutorials on cracking and some on assembly.

http://www.coderz.net
  Very good site for virus related shit. They also host tons of other good sites.
  If you're into or planning on making viruses, check em out.

htt://code.box.sk
  Don't have too much on assembly, but some is better than nothing. Great site for
  other programming related resources though.

http://sennaspy.8m.com/
  Nice site with some cool source code on it. Including DOS 6.22, Quake 1/2/3, and
  various versions of the Award BIOS.

http://www.fastsoftware.com/index.htm
  Has some cool stuff, but is MASM specific. Still worth checking out though

http://www.ice-digga.com/programming/
  A few tutorials, mainly on multimedia. Including SoundBlaster programming and
  2D/3D graphics

http://www.text-files.org/
  Kick ass site by RedPriest from #HackPhreak and Condemned. Stilll under construction
  but already has thousands of text files on everything computer. I'll be uploading
  all my assembly resources there as well. ph33r it!

http://www.coderz.net/29a/29a-home.html
  One of the great sites hosted by Coderz.net. 29A is a virus coding group, and their
  E-Zine is one of the best around, not just for virus writers. To give you a hint,
  issue 8 consists of 8 megs of tutorials.

http://www.mandrag0re.net/
  Great guy who has helped me alot. His site has lots of source code on everything from
  viruses in Linux and DOS to various exploits, all done in pure assembly.

http://bobrich.lexitech.com/
  More assembly shit.

http://www.x86.org/
  Great hardware site. Has lots of shit on undocumented x86 shit, and even CPU exploits

http://www.programmersheaven.com/
  Also a nice site. Their assembly section has lots of tutorials, sample source code,
  and libraries.

Appendix B:
Credits, Contact information, Other shit
==========

Credits

First and foremost I would like to thank all assembly programmers out there who have
helped people like me by sharing what they have discovered in form of books, text

files, and sample source code.

Special thanks to:
cozgedal - For his occasional "kick ass"
rpc   - For always helping me with shit, ph33r him
zcyl   - See previous
#unholy   - It just 0wnz

People who have "beta tested" this thing:
snider
cozgedal

Other cool people who have helped me with various things (some without knowing it):
skin_dot, moJoe, lindex, RedPriest

Contact information

E-Mail : fu@ckz.org
Website: http://awc.rejects.net
ICQ   : 42439352
IRC   : irc.ckz.org in #Security/#Unholy/#Computers

Other shit

If you find a mistake (technical, speling, etc) contact me asap

I need feedback! If you have comments please direct them to the e-mail address above.
Constructive negative comments are welcome, but if you just wanna bitch to me try
e-mailing root@microsoft.com instead. After all, that's what microsoft is there for.

If you made use of this tutorial, please contact me as well. I wanna see what people
have done with this.

If you plan to make commercial use of this tutorial (yeah right), contact me.

If you fuck up your computer as a result of this tutorial, don't blame me. All code
has been tested and works great, but I cannot be held responsible for anything that
happens to you as a result of using this information.

You may freely distribute this text as long as you don't change anything. If there's
something you think should be changed, contact me first.

And finally, I'm already working on the sequel to this tutorial. If you didn't get
enough of assembly, check it out. Might take a while to get done though. It will cover
shit like:
Multi-dimensional Arrays
Structures
Code optimization
Macros
Procedures and functions
Reading and Writing directly to sectors
Protected Mode
Multi-Tasking in DOS (well kinda)
OOP (Object Orientated Paradigm)

Some final words:
The key to mastering assembly is LOTS of practise! Don't worry if you don't understand
half of the stuff I talked about here. Put this thing aside and just make lots and lots
of little programs. If they don't work, debug them, even if that takes you all night or
longer. Than come back to this. And don't bother trying to find help. There are only

very few people who know assembly, and if you can figure it out yourself you learn more.
By the way, 4 months after I first opened up a text file on assembly my tasm directory
contains 93 working .asm files coded by myself. On average that's almost 1 program
a day. Remember, you don't have to start coding something big, as long as you code
something!
Don't expect to learn everything in this tutorial within a few days, I would say that
if you can do it in 4 months or so you are doing great.

Quote of the month:
The only good is knowledge and the only evil ignorance.
                                        - Socrates

EOF