Win32Asm Tutorial

Welcome to the win32asm tutorial. This is the online tutorial in htmlhelp format. The tutorial is always under construction so make sure you've got the latest document.

Win32Asm Tutorial

Contents

Note that these tutorials are continuously under construction.

This is my win32asm tutorial. It is always under construction, but I'm working on it. With the next and prev links in the navigation above, you can go the next and previous page.

Introduction

First a short introduction about this tutorial. Win32asm isn't a very popular programming language, and there are only a few (good) tutorials. Also, most tutorials focus on the win32 part of the programming (i.e. the win API, use of standard windows programming techniques and so on), not on the assembler programming itself, using opcodes, registers etc. Although you can find these things in other tutorials, these tutorials often explain DOS programming. This sure helps you to learn the assembly language, but with programming in windows, you don't need to know about DOS interrupts and port in/out functions. In windows, there's the windows API that supplies standard functions you can use in your program, but more on this later. The goal of this tutorial is to explain win32 programming in assembler as well as assembly language itself.

[top]

1.0 - Assembly Language

Assembly language is created as replacement for the raw binary code that the processor understands. A long time ago, when there were no high-level programming languages yet, programs were created in assembly. Assembly codes directly represent instructions the processor can execute. For example:

```
add eax, edx
```

This instruction, add, adds two values together. Eax and edx are called registers, they can contain values and are stored internally in the processor. This code is converted to 66 03 C2 (hexcodes). The processor reads these codes, and executes the instruction it represents. High level languages like C convert their own language to assembly, and the assembler converts it to binary codes:

C code		Assembly		Raw output (hex)
`a = a + b;`	**>> Compiler > >**	`add eax, edx`	**>>Assembler>>**	`66 03 C2`

(Note that this assembly code is simplified, output depends on the context of the C code)

1.1 - Why?

Why would you use asm instead of C or something if it's harder to program in asm??. Assembler programs are small & fast. In very high-level programming languages like artificial intelligence, it gets harder for the compiler to produce output code. The compiler has to figure out the fastest (or smallest) method to produce assembly code, and although compilers still get better, programming the code yourself (with option code optimalization) will produce smaller and faster code. But of course this is much harder than high-level languages.
There's another difference with some high-level languages, that use runtime dll's for their functions. For example, Visual C++ has msvcrt.dll which contains standard C functions. This works OK most of the time, but sometimes causes problems with dll versions (dll hell) and the user always needs to have these DLL's installed. For visual C this is not really a problem, they are installed with the windows-installation. Visual Basic even doesn't convert it's own language to assembler (although version 5 and above do this a little, but not fully), it depends highly on msvbvm50.dll, the Visual basic virtual machine. The exe that is created by VB exists solely of simple pieces of code and many calls to this dll. This is why VB is slow. Assembler is the fastest language there is. It only uses the system DLL's kernel32.dll, user32.dll, etc.

Another misunderstanding is that many people think of assembly as an impossible language to program in. Sure, it is difficult, but not impossible. Creating big projects in assembly indeed is hard in assembler, I only use it for small programs, or DLL's that can be imported by other languages for parts of the code that need speed. Also, there's a big difference between DOS & windows programs. DOS programs use interrupts as 'functions'. Like int 10 for video, int 13 for file access etc. In win, there's the API, Application Programming Interface. This interface consists of functions you can use in your program. In dos programs, interrupts have an interrupt number and a function number. In win, API functions just have names (e.g. MessageBox, CreateWindowEx). You can import libraries (DLL's) and use the functions inside them. This makes it a lot easier to program in asm. You will learn more about this in the next tutorials.

[top]

2.0 - Getting started

Enough introduction for now, let's get started. To program in assembly, you will need some tools. Below, you can see which tools I will use in this tutorial. I advise you to install the same tools, so you can follow the tutorial and try the examples. I've also given some alternatives, for most tools you can choose the alternative for this tutorial, but be warned that there is a big difference between the assemblers (masm, tasm and nasm). In this tutorial masm will be used, because of it's useful functions like *invoke*, which makes the programming much easier. Of course your free to go and use the assembler you prefer, but it will be harder to follow this tutor and you will have to convert the examples to make it work with your assembler.

Assembler
Used: Masm (from the win32asm package) **Location:** Win32asm.cjb.net **Description:** An assembler converts the assembly source code (opcodes) to the raw output (object file) for the processor. **About:** Masm, macro assembler, is an assembler with a few useful features, like 'invoke', which simplifies calls to API functions and data type checking, but you will understand this later in this tutorial. If you have read the text above you will know that for this tutorials it's advised to use masm.
Alternatives: Tasm, nasm [dl]

Linker
Used: Microsoft Incremental Linker (link.exe) **Location:** Win32asm.cjb.net (in the win32asm package) **Description:** A linker 'links' all object files and libraries (for DLL import) together to produce the final executable. **Description:** I will use link.exe which is available in the win32asm package at Iczelion's page, but most linkers can be used.
Alternatives: Tasm linker

Resource Editor
Used: Borland Resource Workshop **Location:** Not free **Description:** A resource editor is used for creating resources (images, dialogs, bitmaps, menu's). **About:** Most editors will be okay my personal favor goes out to resource workshop but you can use what you want. Note: resource files created with resource workshop sometimes gives problems with resource compiling, if you want to use this editor, you should download tasm as well, which contains brc32.exe for compiling borland-style resources.
Alternatives: Symantec Resource Editor, Resource Builder, many others

Text Editor
Used: Ultraedit **Location:** www.ultraedit.com **Description:** Does a text editor need a description?

About: The choice of a text editor is very personal, I like ultraedit very much. You can download my wordfile for ultraedit, so you will have syntax highlighting for assembler code. But at least choose a text editor that supports syntax highlighting (keywords will automatically be colored), this is VERY useful and it makes your code a lot easier to read and write. Ultraedit also has a function list to go to a specific function in your code quick. [download wordfile here]

Alternatives:
One of the millions of text editors

References
Used: Win32 Programmer's reference **Location:** (search the web) **Description:** You will need a few references with API functions. The most important is "win32 programmer's reference" (win32.hlp). This is a big file, about 24 mb (some versions are 12 but are not complete). In this file, all the functions of the system dll's (kernel, user, gdi, shell etc.) are described. You will at least need this file, other references (sock2.hlp, mmedia.hlp, ole.hlp etc.) are also useful but not necessary.
Alternatives: N/A

2.1 - Installing the tools

Now you've got these tools, install them somewhere. Here are a few important notes:

- Install the masm package on the same drive you plan writing your assembly source files. This ensures the paths to the includes and libraries are correct
- add the bin directory of masm (and tasm) to your path in autoexec.bat and reboot.
- If you've got ultraedit, use the wordfile you can download above and enable the function-listview.

2.2 - Folder for your source

Create a win32 folder (or with any name you like) somewhere (on the same drive as masm), and create a sub-folder for every project you make.

[top]

3.0 - Basics of asm

This tutorials will teach you the basics of assembly language

3.1 - Opcodes

Assembler programs are created with opcodes. An opcode is an instruction the processor can understand. For example:

```
ADD
```

The add instructions adds two numbers together. Most opcodes have operands:

```
ADD eax, edx
```

ADD has 2 operands. In the case of an addition, a source and a destination. It adds the source value to the destination value and then stores the result in the destination. Operands can be of different types: registers, memory locations, immediate values (see below).

3.2 - Registers

There are a few sizes of registers: 8 bit, 16 bit, 32 bit (and some more on a MMX processor). In 16-bit programs, you can only use 16-bit registers and 8 bit registers. In 32-bit programs you can also use 32-bit registers.
Some registers are part of other registers; for example, if EAX holds the value EA7823BBh, here's what the other registers contain.

EAX	EA	78	23	BB
AX	EA	78	23	BB
AH	EA	78	23	BB
AL	EA	78	23	BB

ax, ah, al are part of eax. Eax is a 32-bit register (available only on 386+), ax contains the lower 16 bits (2 bytes) of eax, ah contains the high byte of ax, and al contains the low byte of ax. So ax is 16 bit, al and ah are 8 bit. So, in the example above, these are the values of the registers:

```
eax = EA7823BB (32-bit)
ax = 23BB (16-bit)
ah = 23 (8-bit)
al = BB (8-bit)
```

Example of the use of registers (don't care about the opcodes, just look at the registers and description):

`mov eax, 12345678h`	mov loads a value into a register (note: 12345678h is a hex value because of the 'h' suffix)
`mov cl, ah`	move the high byte of ax (67h) into cl
`sub cl, 10`	substract 10 (dec.) from the value in cl
`mov al, cl`	and store it in the lowest byte of eax.

Let's examine the code above:
The mov instruction can move a value from a register, memory or an immediate value to another register. In the example above, eax contains 12345678h. Then the value of ah (the 3rd byte from the left in eax) is copied to cl (the lowest byte of the ecx register). Then, 10 is substracted from cl and it is moved back to al (the lowest byte of eax).

There are different types of registers:

General Purpose

These 32-bit (and 16/8 for their components) registers can be used for anything:

eax (ax/ah/al)	Accumulator
ebx (bx/bh/bl)	Base
ecx (cx/ch/cl)	Counter
edx (dx/dh/dl)	Data

Although they have names, you can use them for anything.

Segment Registers

Segment registers define the segment of memory that is used. You'll probably won't need them with win32asm, because windows has a flat memory system. In dos, memory is divided into segments of 64kb, so if you want to define a memory address, you specify a segment, and an offset (like 0172:0500 (segment:offset)). In windows, segments have sizes of 4gig, so you won't need segments in win. Segments are always 16-bit registers.

CS	code segment
DS	data segment
SS	stack segment
ES	extra segment
FS (only 286+)	general purpose segment
GS (only 386+)	general purpose segment

Pointer Registers

Actually, you can use pointer registers as general purpose registers (except for eip), as long as you preserve their original values. Pointer registers are called pointer registers because their often used for storing memory addresses. Some opcodes also (movb,scasb,etc.) use them.

esi (si)	Source index
edi (di)	Destination index
eip (ip)	Instruction pointer

EIP (or IP in 16-bit programs) contains a pointer to the instruction the processor is about to execute. So you can't use eip as general purpose registers.

Stack Registers

There are 2 stack registers: esp & ebp. Esp holds the current stack position in memory (more about this in one of the next tutorials). Ebp is used in functions as pointer to the local variables.

esp (sp)	Stack pointer
ebp (bp)	Base pointer

[top]

4.0 - Memory

This section will explain how memory is handled in windows.

4.1 - DOS & win 3.xx

In 16-bit programs like for DOS and windows 3, memory was divided in segments. These segments have sizes of 64kb. To access memory, a segment pointer and an offset pointer are needed. The segment pointer indicates which segment (section of 64kb) to use, the offset pointer indicates the place in the segment itself. Look at the following picture:

MEMORY				
SEGMENT 1 (64kb)	SEGMENT 2 (64kb)	SEGMENT 3 (64kb)	SEGMENT 4(64kb)	and so on

Note that the following explanation is for 16-bit programs, more on 32-bit later (but don't skip this part, it is important to understand 32-bits).
The table above is the total memory, divided in segments of 64kb. There's a maximum of 65536 segments. Now take one of the segments:

SEGMENT 1(64kb)					
Offset 1	Offset 2	Offset 3	Offset 4	Offset 5	and so on

To point to a location in a segment, offsets are used. An offset is a location inside the segment. There's a maximum of 65536 offsets per segment. The notation of an address in memory is:

```
SEGMENT:OFFSET
```

For example:

```
0030:4012 (all hex numbers)
```

This means: segment 30, offset 4012. To see what is at that address, you first go to segment 30, and then to offset 4012 *in* that segment. In the previous tutorials, you've learned about segment and pointer registers. For example, the segment registers are:

```
CS - Code segment
DS - Data Segment
SS - Stack Segment
ES - Extra Segment
FS - General Purpose
GS - General Purpose
```

The names explain their function: code segment (CS) contains the number of the section where the current code that is executed is. Data segment for the current segment to get data from. Stack indicates the stack segment (more on the stack later), ES, FS, GS are general purpose registers and can be used for any segment (not in win32 though).

Pointer registers most of the time hold an offset, but general purpose registers (ax, bx, cx, dx etc.) can also be used for this. IP indicates the offset (in the CS (code segment)) of the instruction that is currently executed. SP holds the offset (in the SS (stack segment)) of the current stack position.

4.2 - 32-bit Windows

You have probably noticed that all this about segments really isn't fun. In 16-bit programming, segments are essential. Fortunately, this problem is solved in 32-bit windows (95 and above). You still have segments, but don't care about them because they aren't 64kb, but **4 GIG**. Windows will probably even crash if you try to change one of the segment registers. This is called the *flat memory model*. There are only offsets, and they now are 32-bit, so in a range from 0 to 4,294,967,295. Every location in memory is indicated only by an offset. This is really one of the best advantages of 32-bit over 16-bit. So you can forget the segment registers now and focus on the other registers.

[top]

5.0 - Opcodes

Opcodes are the instructions for the processor. Opcodes are actually "readable text"-versions of the raw hex codes. Because of this, assembler is the lowest level of programming languages, everything in asm is directly converted to hexcodes. In other words, you don't have a compiler-fase that converts a high-level language to low-level, the assembler only converts assembler codes to raw data.

This tutor will discuss a few opcodes that have to do with calculation, bitwise operations, etc. The other opcodes, jump instructions, compare-opcodes etc, will be discussed later.

5.1 - A few basic calulation opcodes

MOV

This instruction is used to move (or actually copy) a value from one place to another. This 'place' can be a register, a memory location or an immediate value (only as source value of course). The syntax of the **mov** instruction is:

```
mov destination, source
```

You can move a value from one register to another (note that the instruction *copies* the value, in spite of its name 'move', to the destination).

```
mov edx, ecx
```

The instruction above copies the contents of ecx to edx. The size of source and destination should be the same, this instruction for example is NOT valid:

```
mov al, ecx ; NOT VALID
```

This opcode tries to put a DWORD (32-bit) value into a byte (8-bit). This can't be done by the mov instruction (there are other instructions to do this). But these instructions are allowed because source and destination don't differ in size.

```
mov al, bl
mov cl, dl
mov cx, dx
mov ecx, ebx
```

Memory locations are indicated with an offset (in win32, for more info see the previous page). You can also get a value from a certain memory location and put it in a register. Take the following table as example:

offset	34	35	36	37	38	39	3A	3B	3C	3D	3E	3F	40	41	42
data	0D	0A	50	32	44	57	25	7A	5E	72	EF	7D	FF	AD	C7

(each block represents a byte)

The offset value is indicated as a byte here, but it is a 32-bit value. Take for example 3A (which isn't a common value for an offset, but otherwise the table won't fit...), this also is a 32-bit value: 0000003Ah. Just to save space, some unusual and low offsets are used. All values are hexcodes.

Look at offset 3A in the tabel above. The data at that offset is 25, 7A, 5E, 72, EF, etc. To put the value at offset 3A in, for example, a register you use the mov instruction, too:

```
mov eax, dword ptr [0000003Ah]
```

(the h-suffix means hex value)
The instruction **mov eax, dword ptr [0000003Ah]** means: put the value with the size of a DWORD (32-bit) at memory location 3Ah in register eax. After executing this instruction, eax contains the value 725E7A25h. Maybe you have noticed that this is the inverse of what's in the memory: 25 7A 5E 72. This is because values are stored in memory using the **little endian** format. This means that the rightmost byte is stored in the most significant byte: The byte order is reversed. I think some examples will make this clear:

```
the dword (32-bit) value 10203040 hex is stored in memory as: 40, 30, 20,
10 (each value consumes one byte (8-bit))
the word (16-bit) value 4050 hex is stored in memory as 50, 40
```

Back to the example above. You can do this with other sizes too:

```
mov cl, byte ptr [34h] ; cl will get the value 0Dh (see table above)
mov dx, word ptr [3Eh] ; dx will get the value 7DEFh (see table above,
remember the reversed byte order)
```

The size sometimes isn't necessary:

```
mov eax, [00403045h]
```

because eax is a 32-bit register, the assembler assumes (and this is the only way to do it, too) it should take a 32-bit value from memory location 403045hex.

Immediate numbers are also allowed:

```
mov edx, 5006
```

This will just make the register edx contain the value 5006. The brackets, **[** and **]**, are used to get a value from the memory location between the brackets, without brackets it is just a value. A register as memory location is allowed to (it should be a 32-bit register in 32-bit programs):

```
mov eax, 403045h ; make eax have the value 403045 hex.
mov cx, [eax] ; put the word size value at the memory location EAX
(403045) into register CX.
```

In **mov cx, [eax]**, the processor first looks what value (=memory location) *eax* holds, then what value is at that location in memory, and put this word (16 bits because the destination, cx, is a 16-bit register) into CX.

ADD, SUB, MUL, DIV

Many opcodes do calculations. You can guess most of their names: add (addition), sub (substraction), mul (multiply), div (divide) etc.

The *add*-opcode has the following syntax:

```
add destination, source
```

The calculation performed is *destination* = *destination* + *source*. The following forms are allowed:

Destination	Source	Example
Register	Register	`add ecx, edx`
Register	Memory	`add ecx, dword ptr [104h] / add ecx, [edx]`
Register	Immediate value	`add eax, 102`
Memory	Immediate value	`add dword ptr [401231h], 80`

Memory Register `add dword ptr [401231h], edx`

This instruction is very simple. It just takes the source value, adds the destination value to it and then puts the result in the destination. Other mathematical instructions are:

```
sub destination, source (destination = destination - source)
mul destination, source (destination = destiantion * source)
div source (eax = eax / source, edx = remainer)
```

Substraction works the same as add, multiplication is just dest = dest * source. Division is a little different. Because registers are integer values (i.e. round numbers, not floating point numbers) , the result of a division is split in a *quotient* and a *remainder*. For example:

```
28 /6 --> quotient = 4, remainder = 4
30 /9 --> quotient = 3, remainder = 3
97 / 10 --> quotient = 9, remainder = 7
18 /6 --> quotient = 3, remainder = 0
```

Now, depending on the size of the source, the quotient is stored in (a part of) eax, the remainder in (a part of) edx:

Source size	Division	Quotient stored in ...	Remainder Stored in...
BYTE (8-bits)	**ax / source**	AL	AH
WORD (16-bits)	**dx:ax* / source**	AX	DX
DWORD (32-bits)	**edx:eax* / source**	EAX	EDX

* = For example: if dx = 2030h, and ax = 0040h, dx: ax = 20300040h. dx:ax is a dword value where dx represents the higher word and ax the lower. Edx:eax is a quadword value (64-bits) where the higher dword is edx and the lower eax.

The source of the div-opcode can be:

- an 8-bit register (al, ah, cl,...)
- a 16-bit register (ax, dx, ...)
- a 32-bit register (eax, edx, ecx...)
- an 8-bit memory value (byte ptr [xxxx])
- a 16-bit memory value (word ptr [xxxx])
- a 32-bit memory value (dword ptr [xxxx])

The source can **not** be an immediate value because then the processor cannot determine the size of the source operand.

BITWISE OPERATIONS

These instructions all take a destination and a source, exept the 'NOT' instruction. Each bit in the destination is compared to the same bit in the source, and depending on the instruction, a 0 or a 1 is placed in the destination bit:

Instruction	AND				OR				XOR				NOT	
Source Bit	0	0	1	1	0	0	1	1	0	0	1	1	0	1
Destination Bit	0	1	0	1	0	1	0	1	0	1	0	1	X	X
Output Bit	0	0	0	1	0	1	1	1	0	1	1	0	1	0

AND sets the output bit to 1 if *both* the source and destination bit is 1.
OR sets the output bit if *either* the source or destination bit is 1
XOR sets the output bit if the source bit is different from the destination bit.
NOT inverts the source bit.

An example:

```
mov ax, 3406
mov dx, 13EAh
xor ax, dx
```

ax = 3406 (decimal), which is 0000110101001110 in binary.
dx = 13EA (hex), which is 0001001111101010 in binary.
Perform the XOR operation on these bits:

Source	0001001111101010 (dx)
Destination	0000110101001110 (ax)
Output	0001111010100101 (new dx)

The new dx is 0001111010100101 (7845 decimal, 1EA5 in hex) after the instruction.

Another example:

```
mov ecx, FFFF0000h
not ecx
```

FFFF0000 is in binary 11111111111111110000000000000000 (16 1's, 16 0's)
If you take the inverse of every bit, you get:
00000000000000001111111111111111 (16 0's, 16 1's), which is 0000FFFF in hex.
So ecx is after the NOT operation 0000FFFFh.

IN/DECREMENTS

There are 2 very simple instructions, DEC and INC. These instructions increase or decrease a memory location or register with one. Simply put:

```
inc reg -> reg = reg + 1
dec reg -> reg = reg - 1
inc dword ptr [103405] -> value at [103405] will increase by one.
dec dword ptr [103405] -> value at [103405] will decrease by one.
```

NOP

This instruction does absolutely nothing. This instruction just occupies space and time. It is used for filling purposes and patching codes.

Bit Rotation and Shifting

Note: Most of the examples below use 8-bit numbers, but this is just to make the picture clear.

Shifting functions

SHL destination, count
SHR destination, count

SHL and SHR shift a *count* number of bits in a register/memlocation left or right.

Example:

```
; al = 01011011 (binary) here
shr al, 3
```

This means: shift all the bits of the al register 3 places to the right. So al will become 00001011. The bits on the left are filled up with zeroes and the bits on the right are shifted out. The last bit that is shifted out is saved in the *carry-flag*. The carry-bit is a bit in the processor's *Flags register*. This is not a

register like eax or ecx that you can directly access (although there are opcodes to do this), but it's contents depend on the result of the instruction. This will be explained later, the only thing you'll have to remember now is that the carry is a bit in the flag register and that it can be on or off. This bit equals the last bit shifted out.

shl is the same as *shr*, but shifts to the left.

```
; bl = 11100101 (binary) here
shl bl, 2
```

bl is 10010100 (binary) after the instruction. The last two bits are filled up with zeroes, the carry bit is 1, because the bit that was last shifted out is a 1.

Then there are two other opcodes:

SAL destination, count (Shift Arithmetic Left)
SAR destination, count (Shift Arithmetic Right)

SAL is the same as SHL, but SAR is not quite the same as SHR. SAR does not shift in zeroes but copies the MSB (most significant bit). Example:

```
al = 10100110
sar al, 3
al = 11110100
sar al, 2
al = 11111101

bl = 00100110
sar bl, 3
bl = 00000010
```

Rotation functions

rol destination, count ; rotate left
ror destination, count ; rotate right
rcl destination, count ; rotate through carry left
rcr destination, count ; rotate through carry right

Rotation looks like shifting, with the difference that the bits that are shifted out are shifted in again on the other side:

Example: ror (rotate right)

	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0	
Before	1	0	0	1	1	0	1	1	
Rotate, count= 3				1	0	0	1	1	**0 1 1** (Shift out)
Result	1	1	0	1	0	0	1	1	

As you can see in the figure above, the bits are rotated, i.e. every bit that is pushed out is shift in again on the other side. Like shifting, the carry bit holds the last bit that's shifted out. RCL and RCR are actually the same as ROL and RCR. Their names suggest that they use the carry bit to indicate the last shift-out bit, which is true, but as ROL and ROR do the same, they do not differ from them.

Exchange

The XCHG instruction is also quite simple. It can exchange two registers or a register and a memory location:

```
eax = 237h
```

```
ecx = 978h
xchg eax, ecx
eax = 978h
ecx = 237h
```

[top]

6.0 - File Structure

Assembly source files are divided in sections. The sections are code, data, uninitialized data, constants, resource and relocations. Resource sections are created by a resource file, more about this later. The relocation section is not important to us (it contains information to make it possible for the PE-loader to load the program at a different location in memory). Important sections are code, data, uninitialized data and constants. Code sections contain, well you've guessed it, code. Data contains data, and has read and write access. The whole data section is included in the exe file and can be initialized with data.
Unitialized data has no contents at startup, and isn't even included in the exe file itself, it is just a part of memory reserved by windows. This section has read and write access. Constants is the same as the data section, but with readonly access. Although this section can be used for constants, it is easier and faster to just declare constants in include files, and then use it as immediate values.

6.1 Section indicators

In your source files (*.asm), you define sections with the section statements:

.code ; code section starts here
.data ; data section starts here
.data? ; unitialized data starts here
.const ; constants section starts here

Executable files (*.exe, *.dll and more) are (in win32) in the portable executable-format (PE). I won't discuss this in details but a few things are important. The sections are defined in the PE-header with a few characteristics:
Section name, RVA, offset, raw size, virtual size and flags. RVA (relative virtual address) is the *relative* location in memory where that section will be loaded. Relative here means that it is relative to the *base address*, the address in memory where the program is loaded. This address is also in the PE-header but can be changed by the PE-loader (using the relocation-section). Offset is the raw offset in the exe file itself where the initial data is. Virtual size is the size it will become in memory. Flags are the flags for read-access/write-access/executable etc.

6.2 Example program

Here's an example program:

```
.data
Number1 dd 12033h
Number2 dw 100h,200h,300h,400h
Number3 db "blabla",0

.data?
Value dd ?

.code
mov eax, Number1
mov ecx, offset Number2
add ax, word ptr [ecx+4]
mov Value, eax
```

This program will not assemble well, but that doesn't matter.
In your assembly program, everything you put in a section will go to the exe file and, when the program is loaded in memory, at a certain memory location. In the data section above, there are 3 labels: Number1, Number2, Number3. These labels will hold the offset of the location they are in the

program so you can use them to indicate a place in your program.
DD directly puts a dword at that place, DW a word and DB a byte. With db, you can also use a string, because a string is actually a list of byte values. In the example, the data section would become this in memory:
33,20,01,00,00,01,00,02,00,03,00,04,62,6c,61,62,6c,61,00 (all hex numbers)
(every value is a byte)

I've colored some of the numbers. Number1 points to the memory location where the byte 33 is, Number 2 points to the location of the red 00, Number3 to the green 62. Now if you use this in your program:

```
mov ecx, Number1
```

It actually means:

```
mov ecx, dword ptr [location where the dword 12033h is in memory]
```

But this:

```
mov ecx, offset Number1
```

means:

```
mov ecx, location where dword 12033h is in memory
```

In the first example, ecx will get the value that is at the memory location of Number1. In the second, ecx will become the *memory location (offset) itself*. These two examples below have the same effect:

(1)
```
mov ecx, Number1
```

(2)
```
mov ecx, offset Number1
mov ecx, dword ptr [ecx]
```
(or mov ecx, [ecx])

Now let's go back to the example:

```
.data
Number1 dd 12033h
Number2 dw 100h,200h,300h,400h
Number3 db "blabla",0

.data?
Value dd ?

.code
mov eax, Number1
mov ecx, offset Number2
add ax, word ptr [ecx+4]
mov Value, eax
```

The label Value can be used just like Number1, Number2 and Number3, but it will contain 0 at startup because it is in the unitialized data section. The advantage of this is, that everything you define in .data? will not be in the executable, only in memory.

```
.data?
ManyBytes1 db 5000 dup (?)

.data
ManyBytes2 db 5000 dup (0)
```

(5000 dup means: 5000 duplicates. *Value db 4,4,4,4,4,4,4* is the same as *Value db 7 dup (4)*.)

ManyBytes1 will not be in the file itself, just 5000 reserved bytes in memory. But ManyBytes2 will be in the executable, making the file 5000 bytes bigger. As your file then will contain 5000 zeroes, this is not very useful.

The code section will just be assembled (converted to raw codes) and placed in the executable (and in memory when it's loaded of course).

[top]

7.0 - Conditional Jumps

In the code section, you can also use labels like this:

```
.code

mov eax, edx
sub eax, ecx
cmp eax, 2
jz loc1
xor eax, eax
jmp loc2
loc1:
xor eax, eax
inc eax
loc2:
```

(xor eax, eax means: eax = 0.)

```
Let's examine the code:
mov eax, edx : put edx in eax
sub eax, ecx : substract ecx from eax
cmp eax, 2
```

This is a new instruction: `cmp`. `Cmp` stands for compare. It can compare two values (reg, mem, imm) and sets the Z-flag (zeroflag) if they are equal. The zero-flag is, like the carry, also a bit in the internal flag register.

```
jz loc1
```

This one is also new. It is a conditional jump. `Jz` = jump if zero. I.e. jump if the zero flag is set. Loc1 is a label for the offset in memory where the instructions '`xor eax, eax | inc eax`' begin. So `jz loc1` = jump to the instruction at loc1 if the zero flag is set.

```
cmp eax, 2 : set zero flag if eax=2
jz loc1 : jump if zero flag is set
```

=

```
Jump to the instructions at loc1 if eax is equal to 2
```

Then there's a `jmp loc2`. This also is a jump, but an unconditional jump: it always jumps. What the code above exactly does is:

```
if ((edx-ecx)==2)
{
eax = 1;
}
else
{
eax = 0;
}
```

or the BASIC version:

```
IF (edx-ecx)=2 THEN
EAX = 1
ELSE
```

```
EAX = 0
END IF
```

7.1 - Flag register

The flag register has a set of flags which are set or unset depending on calculations or other events. I won't discuss all of them, only a few are important:

ZF (Zero flag)
This flag is set when the result of a calculation is zero (compare is actually a substraction without saving the results, but setting the flags only).
SF (Sign flag)
If set, the resulting number of a calculation is negative.
CF (Carry flag)
The carry flag contains the left-most bit after calculations.
OF (Overflow flag)
Indicates an overflow of a calculation, i.e. the result does not fit in the destination.

There are more flags (Parity, Auxiliary, Trap, Interrupt, Direction, IOPL, Nested Task, Resume, & Virtual Mode) but as we won't use them I don't explain them.

7.2 - The jump serie

There is a whole serie of conditional jumps, and they all jump depending on the state of the flag. But as most jumps have clear names, you don't even have to know which flag has to be set. `"Jump if greater or equal"` (jge) for example is the same as `"Sign flag = Overflow flag"`, and `"Jump if zero"` is the same as `"Jump if Zero flag = 1"`.

In the table below, 'meaning' indicates what the outcome of a calculation should be to jump. "Jump if above" means:
cmp x, y
jump if x is above y

Opcode	Meaning	Condition
JA	Jump if above	CF=0 & ZF=0
JAE	Jump if above or equal	CF=0
JB	Jump if below	CF=1
JBE	Jump if below or equal	CF=1 or ZF=1
JC	Jump if carry	CF=1
JCXZ	Jump if CX=0	register CX=0
JE (is the same as JZ)	Jump if equal	ZF=1
JG	Jump if greater (signed)	ZF=0 & SF=OF
JGE	Jump if greater or equal (signed)	SF=OF
JL	Jump if less (signed)	SF != OF
JLE	Jump if less or equal (signed)	ZF=1 or SF!=OF
JMP	Unconditional Jump	-
JNA	Jump if not above	CF=1 or ZF=1
JNAE	Jump if not above or equal	CF=1
JNB	Jump if not below	CF=0
JNBE	Jump if not below or equal	CF=1 & ZF=0
JNC	Jump if not carry	CF=0
JNE	Jump if not equal	ZF=0
JNG	Jump if not greater (signed)	ZF=1 or SF!=OF

JNGE	Jump if not greater or equal (signed)	SF!=OF
JNL	Jump if not less (signed)	SF=OF
JNLE	Jump if not less or equal (signed)	ZF=0 & SF=OF
JNO	Jump if not overflow (signed)	OF=0
JNP	Jump if no parity	PF=0
JNS	Jump if not signed (signed)	SF=0
JNZ	Jump if not zero	ZF=0
JO	Jump if overflow (signed)	OF=1
JP	Jump if parity	PF=1
JPE	Jump if parity even	PF=1
JPO	Jump if paity odd	PF=0
JS	Jump if signed (signed)	SF=1
JZ	Jump if zero	ZF=1

All jump instructions take one operand: an offset to jump to.

[top]

8.0 - Something about numbers

Using integer or floating point numbers in most programming languages just depends on the declaration of variables. In assembler these are completly different. Floating point calculations are done with a special opcode-set, by a FPU co-processor (floating point unit). Floating point instructions will be discussed later. First something about integers. In C language, there are signed and unsigned numbers. Signed just means that the number has a sign (+ or -), unsigned is always positive. Look at the table below to see the difference (again, this is a byte-example, it works the same for other sizes):

Value	00	01	02	03	...	7F	80	...	FC	FD	FE	FF
Unsigned meaning	00	01	02	03	...	7F	80	...	FC	FD	FE	FF
Signed meaning	00	01	02	03	...	7F	-80	...	-04	-03	-02	-01

So with signed numbers, a byte is split in two ranges: 0 - 7F for positive values, 80 - FF for negative values. For dword values, it works the same: 0 - 7FFFFFFFh positive, 80000000 - FFFFFFFFh negative. As you might have noticed, negative numbers have the most significant bit set, because they are greater that 80000000h. This bit is called the sign bit.

8.1 - Signed or unsigned?

Neither you or the processor can see if a value is singed or unsigned. The good news is that for addition and substraction, it doesn't matter if the number is signed or unsigned:

```
Calculate: -4 + 9
FFFFFFFC + 00000009 = 00000005. (which is correct)

Calculate 5 - (-9)
00000005 - FFFFFFF7 = 0000000E (which is correct, too ( 5 - -9 = 14)
```

The bad news is that this is not true for muliplication, division and compares. Therefore, there are special mul and div opcodes for signed numbers:

`imul` and `idiv`

Imul also has an advantages over mul because it can take immediate values:

```
imul src
imul src, immed
imul dest,src, 8-bit immed
imul dest,src

idiv src
```

They are about the same as mul and div, but they calculate with signed values. Compares can be used the same as with unsigned numbers, but the flags are set different. Therefore there are different jump instructions for signed and unsigned numbers:

```
cmp ax, bx
ja somewhere
```

Ja is an unsigned jump. Jump if above. Imagine that ax = FFFFh (FFFFh unsigned, -1 signed) and bx = 0005h (5 unsigned, 5 signed). As FFFFh is a higher (unsigned) value than 0005, the ja-instruction will jump. But if the jg instruction is used (which is a signed jump):

```
cmp ax, bx
jg somewhere
```

The jg-instruction won't jump, because -1 is not greater that 5.

Just remember this:
A number is signed or unsigned depending on how you treat the number.

[top]

9.0 - More opcodes

Here are some more opcodes

TEST

Test performs a logical AND operation on the two operands (dest, src) and sets the flag register according to the result. The result itself is not stored. Test is used to test for a bit in for example a register:

```
test eax, 100b ; (b-suffix stands for binary)
jnz bitset
```

The jnz will jump if the 3rd bit from the right in eax is set. A very common use of test is to test if a register is zero:

```
test ecx, ecx
jz somewhere
```

The jz jumps if ecx is zero.

STACK OPCODES

Before I will tell you about the stack opcodes, I will first explain what the stack is. The stack is a place in memory, pointed to by the stack pointer register, esp. The stack is a place to hold temporary values. There are two instructions to put a value and get it again: push and pop. Push pushes a value onto the stack, pop pops it off again. The value that was last put on the stack, goes first off. As a value is placed on the stack, the stack pointer decreases, when it is removed, the stack pointer increases. Look at this example:

```
(1) mov ecx, 100
(2) mov eax, 200
(3) push ecx ; save ecx
(4) push eax
(5) xor ecx, eax
(6) add ecx, 400
(7) mov edx, ecx
(8) pop ebx
(9) pop ecx
```

Explaination:

```
1: put 100 in ecx
2: put 200 in eax
3: push ecx (=100) onto the stack (first pushed)
4: push eax (=200) onto the stack (last pushed)
5/6/7: perform operation on ecx, value of ecx changes
8: pop ebx: ebx will become 200 (last pushed, first pop-ed)
9: pop ecx: ecx will become 100 again (first pushed, last pop-ed)
```

To indicate what happens in memory with this pushing and poping, look at the following figures:

Offset	1203	1204	1205	1206	1207	1208	1209	120A	120B
Value	00	00	00	00	00	00	00	00	00
					ESP				

(the stack is here initialy filled with zeroes, but in reality it is not like this. ESP indicates the offset that ESP points to)

```
mov ax, 4560h
push ax
```

Offset	1203	1204	1205	1206	1207	1208	1209	120A	120B
Value	00	00	**60**	**45**	00	00	00	00	00
			ESP						

```
mov cx, FFFFh
push cx
```

Offset	1203	1204	1205	1206	1207	1208	1209	120A	120B
Value	**FF**	**FF**	**60**	**45**	00	00	00	00	00
	ESP								

```
pop edx
```

Offset	1203	1204	1205	1206	1207	1208	1209	120A	120B
Value	FF	FF	60	45	00	00	00	00	00
					ESP				

edx is 4560FFFFh now.

CALL & RET

A call jumps to some code and returns as soon as it found a ret-instruction. You can see them as functions or subs in other programming languages. Example:

```
..code..
call 0455659
..more code..

Code at 455659:
add eax, 500
mul eax, edx
ret
```

When the call instruction is executed, the processor jumps to the code at 455659, executes the instructions until the ret, and then returns to the instruction after the call. The code the call jumps to is called a procedure. You can put code that you use many times into a procedure and then use a call each time you need it.

More into details: A call pushes the EIP (pointer to the next instruction that has to be executed) on the stack, and the ret-instruction pops it off again and returns. You can also specify arguments for a call. This is done by pushes:

```
push something
push something2
call procedure
```

Inside the call, the arguments can be read from the stack and used. Local variables, i.e. data that is only needed within the procedure, are also stored on the stack. I won't go into details about this, because it can be done very easily with masm and tasm. Just remember you can make procedures, and that they can use parameters. One important note:

eax is almost always used to hold the return value of a procedure

This is also true for windows functions. Of course you can use any other register in your own procedure, but this is the standard.

[top]

10.0 - The advantages of masm

If you are not using masm, you can skip this section and try to convert all the examples, or read it anyway and try to persuade yourself using masm. Of course this is your own choice. But masm makes assembly language really easier.

10.1 - Comparison and loop construction

Masm has some pseudo-high level syntax to create compare- and loopconstructions easily:

.IF, .ELSE, .ELSEIF, .ENDIF
.REPEAT, .UNTIL
.WHILE, .ENDW, .BREAK
.CONTINUE

If

If you have any experience with programming languages (you should), you probably have seen something like an if/else construction:

```
.IF eax==1
;eax is one
.ELSEIF eax=3
; eax is three
.ELSE
; eax is not one or three
.ENDIF
```

This construction is VERY useful. You don't have to mess with jumps, just an .IF statement (don't forget the period before .IF and .ELSE etc.). Nested if's are allowed:

```
.IF eax==1
.IF ecx!=2
; eax= 1 and ecx is not 2
.ENDIF
.ENDIF
```

But this can be done easier:

```
.IF (eax==1 && ecx!=2)
; eax = 1 and ecx is not 2
.ENDIF
```

These are the operators you can use:

==	is equal to
!=	is not equal to
>	is greater than
<	is less than
>=	is greater than or equal to
<=	is less than to equal to
&	bit-test
!	logical NOT
&&	logical AND

||	logical OR
CARRY?	carry bit set
OVERFLOW?	overflow bit set
PARITY	parity bit set
SIGN?	sign bit set
ZERO?	zero bit set

Repeat

This statement executes a block until a condition is true:

```
.REPEAT
; code here
.UNTIL eax==1
```

This code repeats the code between repeat and until, until eax = 1.

While

The while is the inverse of the repeat function. It executes a block *while* a condition is true:

```
.WHILE eax==1
; code here
.ENDW
```

You can use the `.BREAK` statement to 'break' out of the loop

```
.WHILE edx==1
inc eax
.IF eax==7
.BREAK
.ENDIF
.ENDW
```

If eax=7, the while-loop will stop.

The `continue` instruction makes the repeat or while-block skip to the code that evaluates the condition of the loop.

10.2 - Invoke

This is the biggest advantage over tasm and nasm. Invoke simplifies the use of procedures and calls.

Normal style:

```
push parameter3
push parameter2
push parameter1
call procedure
```

Invoke style:

```
invoke procedure, parameter1, parameter2, parameter3
```

The assembled code is exactly the same, but the invoke style is easier and more reliable. To use invoke on a procedure, you'll have to define a prototype:

```
PROTO STDCALL testproc:DWORD, :DWORD, :DWORD
```

This declares a procedure, named testproc, that takes 3 DWORD-size parameters. Now if you do this...

```
invoke testproc, 1, 2, 3, 4
```

...masm will give you an error that the testproc procedure takes 3 parameters, not 4. Masm also has type checking, i.e. it checks if the parameters have the right type (size).

In an invoke statement, you can use ADDR in stead of OFFSET. This will make an address in the correct form when it's assembled.

Procedures are defined like this:

```
testproc PROTO STDCALL :DWORD, :DWORD, :DWORD

.code

testproc proc param1:DWORD, param2:DWORD, param3:DWORD

ret
testproc endp
```

This creates a procedure named testproc, with three parameters. The prototype is used by invoke calls.

```
testproc PROTO STDCALL :DWORD, :DWORD, :DWORD

.code

testproc proc param1:DWORD, param2:DWORD, param3:DWORD

mov ecx, param1
mov edx, param2
mov eax, param3
add edx, eax
mul eax, ecx

ret
testproc endp
```

Now, the procedure does the following calculation. testproc(param1, param2, param3) = param1 * (param2 + param3). The result, the return value, is stored in eax. Local variables are defined like this:

```
testproc proc param1:DWORD, param2:DWORD, param3:DWORD
LOCAL var1:DWORD
LOCAL var2:BYTE

mov ecx, param1
mov var2, cl
mov edx, param2
mov eax, param3
mov var1, eax
add edx, eax
mul eax, ecx
mov ebx, var1
.IF bl==var2
xor eax, eax
.ENDIF

ret
testproc endp
```

You can't use these variables outside the procedure. They are stored on the stack and removed when the procedure returns.

10.3 - Macro's

Macro's will not be explained now. Maybe in a later tutorial, but right now they are not important to us.

[top]

11.0 - Basics of assembly in windows

Now you have some basic knowledge about assembly language, you will learn how to use assembly in windows.

11.1 - API

The fundamental of programming in windows lies in the windows API, Application Programming Interface. This is a set of functies supplied by the operating system. Every windows program uses these functions. These functions are in the system dll's like kernel, user, gdi, shell, advapi, etc. There are two types of functions: **ANSI** and **Unicode**. This has to do with the way strings are stored. With ansi, each byte represents a symbol (asci-code) and uses a 0-byte to indicate the end of a string (null-terminated). Unicode uses the widechar format, which uses 2 bytes per symbol. This allowes the usage of languages that need more characters like chinees. Widechar strings are terminated with 2 0-bytes. Windows supports both types by using different function names for ansi and unicode. For example:

```
MessageBoxA (A-suffix for ansi)
MessageBoxW (W-suffix for widechar (unicode))
```

We will only use the ansi type.

11.1 - importing dll's

In order to use the functions from the windows API, you need to import the dll's. This is done by import libraries (.lib). These libraries are necessary because they allow the system (windows) to load the dll dynamically, i.e. at dynamic base addresses in memory. In the win32asm package (win32asm.cjb.net) libraries for most standard dll's are supplied. You can load a library with the `includelib` statement of masm.

```
includelib C:\masm32\lib\kernel32.lib
```

This will load the library kernel32.lib. In the examples, this form is used:

```
includelib \masm32\lib\kernel32.lib
```

Now you will see why your assembly source files should be on the same drive as masm. Now you can compile your program on an other computer without changing all the paths to the correct drive.

But an include library is not the only thing you need. An include file (.inc) is also needed. These can be automatically generated from the libraries using the l2inc utiliy. An include file is loaded like this:

```
include \masm32\include\kernel32.inc
```

Inside the include file, the prototypes for the functions in the dll are defined, so you can use invoke.

```
kernel32.inc:
...
MessageBoxA proto stdcall :DWORD, :DWORD, :DWORD, :DWORD
MessageBox textequ <MessageBoxA>
...
```

You can see that the include file contains the ANSI functions and also defines the function names withouth the 'A' to be the same as the real function name: you can use MessageBox in stead of MessageBoxA. After you've included a library and an include file, you can use the function:

invoke MessageBox, NULL, ADDR MsgText, ADDR MsgTitle, NULL

11.2 - Windows include file

Then there's a special include file, called windows.inc most of the time, which contains all constants and structures for the windows API. For example, a message box can have different styles. The fourth parameter of the function is the style. NULL stands for MB_OK, which is just an OK button. the windows include file contains definitions for these styles:

```
MB_OK equ 0
MB_OKCANCEL equ ...
MB_YESNO equ ...
```

So you can use these names as constants:

```
invoke MessageBox, NULL, ADDR MsgText, ADDR MsgTitle, MB_YESNO
```

The example will use the include file from the masm package:

```
include \masm32\include\windows.inc
```

11.3 - Frame

```
.486
.model flat, stdcall

option casemap:none

includelib \masm32\lib\kernel32.lib
includelib \masm32\lib\user32.lib
includelib \masm32\lib\gdi32.lib

include \masm32\include\kernel32.inc
include \masm32\include\user32.inc
include \masm32\include\gdi32.inc
include \masm32\include\windows.inc

.data

blahblah

.code

start:

blahblah

end start
```

This is the basic frame for a windows assembly source file (.asm).

.486	Tells the assembler that it should generate opcodes for a 486 processor (or higher). You can use .386, too but 486 works most of the time.
.model flat, stdcall	Use the flat memory model (discussed in one of the previous tutorials) and use the stdcall calling conventenion. This means that parameters for the function are pushed right to left (last parameter first pushed) and that the function should correct the stack itself when finished. This is standard for almost all windows api functions and dll's.
option casemap:none	Controls the mapping of characters to uppercase. For the windows.inc file to work properly, this should be 'none'.
includelib	discussed above
include	also discussed above

.data	start of the data section (see previous tutorials)
.code	start of the code section (see previous tutorials)
start: end start	label that indicates the start of the program. Not that it doesn't need to be called 'start'. You can use any name for it as long as you indicate that it is a start label using the 'end' statement: startofprog: end startofprog

[top]

12.0 - First Program

It's time to create your first program. Instructions in this tutorial are formatted *like this*.

12.1 - Step 1

If all's okay, you should have a win32 (or win32asm) folder on your harddisk on the same drive as masm. For each project, you should create a subdirectory.

Create a subdirectory called 'Firstprogram' in your win32 dir. Create a new textfile and rename it to 'first.asm'.

important: If you are using ultraedit, make sure you've installed my wordfile and switch on the 'functions' window (view, views/lists, function list). Finally, make sure you have set the tab stop value to 4 spaces (Advanced, configuration, Edit tab)

12.2 - Step 2

Put the following code into first.asm:

```
.486
.model flat, stdcall

option casemap:none

includelib \masm32\lib\kernel32.lib
includelib \masm32\lib\user32.lib

include \masm32\include\kernel32.inc
include \masm32\include\user32.inc
include \masm32\include\windows.inc
```

For now, the only two dll's we need are kernel32 and user32.

12.3 - Step 3

We are going to make the famous 'Hello world' program. To display the 'hello world' string we will use a message box. A messagebox is created using the MessageBox function. You can look up this function in the win32 programmer's reference (see tutor 2) . Here is what it says:

The MessageBox function creates, displays, and operates a message box. The message box contains an application-defined message and title, plus any combination of predefined icons and push buttons.

```
int MessageBox(
HWND hWnd, // handle of owner window
LPCTSTR lpText, // address of text in message box
LPCTSTR lpCaption, // address of title of message box
UINT uType // style of message box
);
```

Parameters

hWnd	Identifies the owner window of the message box to be created. If this parameter is NULL, the message box has no owner window.
lpText	Points to a null-terminated string containing the message to be displayed.

lpCaption	Points to a null-terminated string used for the dialog box title. If this parameter is NULL, the default title Error is used.
uType	Specifies a set of bit flags that determine the contents and behavior of the dialog box. This parameter can be a combination of flags from the following groups of flags.

[--SNIP--]

After this text a whole list of constants and flags (which are defined in windows.inc) follows. I haven't displayed it here because it is quite long. Looking at the reference, you can see that the MessageBox function takes 4 parameters: A parent window, a pointer to a message string, a pointer to a title string and the type of messagebox.

hWnd can be NULL, because our program does not have a window.
lpText has to be a pointer to our text. This just means that the parameter is the offset of the memory location where our text is.
lpCaption is the offset of the title string.
uType is a combination of the values explained in the reference like MB_OK, MB_OKCANCEL, MB_ICONERROR, etc.

Let's first define two strings for the messagebox:

add this to first.asm:

```
.data

MsgText db "Hello world!",0
MsgTitle db "This is a messagebox",0
```

`.data` indicates the start of the data section. With `db` bytes are directly inserted, and as a string is just a collection of bytes, the data section will contain the strings above, 0-terminated by the additional `,0`. MsgText holds the offset of the first string, MsgTitle the offset of the second string. Now we can use the function:

```
invoke MessageBox, NULL, offset MsgText, offset MsgTitle, NULL
```

But because invoke is used, you can use the (more safe) ADDR in stead of offset:

```
invoke MessageBox, NULL, ADDR MsgText, ADDR MsgTitle, NULL
```

We haven't looked at the last parameter yet, but this will work fine because MB_OK (style for a messagebox with an OK button) equals 0 (NULL). But you can use any other style. The uType (4th parameter) definiton is:

Specifies a set of bit flags that determine the contents and behavior of the dialog box. This parameter can be a combination of flags from the following groups of flags.

Now take for example that we want a simple messagebox with an OK button with the 'information'-icon. MB_OK is the style for the OK button, MB_ICONINFORMATION is the style for an information icon. Styles are combined with the 'or' operator. This is **not** the or-opcode. Masm will perform the or operation before assembling. In stead of or, you can use the + sign (addition), but sometimes this gives problems with overlapping styles (one style containg some other style). But in this case you could also have used a +.

```
.code

start:

invoke MessageBox, NULL, ADDR MsgText, ADDR MsgTitle, MB_OK +
MB_ICONINFORMATION

end start
```

Add the code above to your first.asm file

We've added a start label, too. If you would assemble your program now and run it, it would display the messagebox and probably crash after you've clicked ok. This is because the program is not ended, and the processor starts to execute whatever there is after the messagebox code. Programs in windows are ended with ther ExitProcess function:

The ExitProcess function ends a process and all its threads.

```
VOID ExitProcess(

UINT uExitCode // exit code for all threads
);
```

We can use 0 as exit code:

Change your code to this:

```
.code

start:

invoke MessageBox, NULL, ADDR MsgText, ADDR MsgTitle, MB_OK +
MB_ICONINFORMATION
invoke ExitProcess, NULL

end start
```

12.4 - Step 4

So our final program is:

```
.486
.model flat, stdcall

option casemap:none

includelib \masm32\lib\kernel32.lib
includelib \masm32\lib\user32.lib

include \masm32\include\kernel32.inc
include \masm32\include\user32.inc
include \masm32\include\windows.inc

.data
MsgText db "Hello world!",0
MsgTitle db "This is a messagebox",0

.code

start:
invoke MessageBox, NULL, ADDR MsgText, ADDR MsgTitle, MB_OK or
MB_ICONINFORMATION
invoke ExitProcess, NULL

end start
```

12.5 - Step 5

Now we will create an executable from this source code.

*Create a new text file named **make.bat** with the following content:*

```
@echo off
```

```
ml /c /coff first.asm
link /subsystem:windows first.obj
pause>nul
```

Explanation:

ml /c /coff first.asm	ml is the macro assembler (masm). Masm will create the raw code from the program. The options mean: /c = Assemble without linking. (because we use link.exe for this) /coff = generate COFF format object file. This is the standard for a windows executable. first.asm = assemble the file first.asm
link /subsystem:windows first.obj	The linker takes the object file and links it to all the imported dll's and libraries. Options: /subsystem:windows = Create an executable for windows. first.obj = link first.obj

If you've done verything right and run the batch file, there will be a first.exe now. Run it to see the results.

[top]

12.0 - Windows in windows

In this tutorial, we will create a program with a window.

12.1 - Windows

You can probably guess why windows is called windows. In windows, there are two types of programs: GUI applications and console applications. Console mode programs look like DOS programs, they run in a DOS-like box. Most program's you use are GUI (graphical user interface) applications. They have a graphical interface to interact with the user. This is done by creating windows. Almost everything you see in windows is a window. First you create a parent window, and then its client windows (controls) like edit boxes, static controls, buttons etc.

12.2 - Window Classes

Each window has a class name. For your parent window you define your own class. For controls, you can use the standard windows class names (e.g. 'EDIT', 'STATIC', 'BUTTON').

12.3 - Structures

A window class in your program is registered by using the 'RegisterClassEx' function (RegisterClassEx is the extended version of RegisterClass, which is not being used much). The declaration of this function is:

```
ATOM RegisterClassEx(

CONST WNDCLASSEX *lpwcx // address of structure with class data
);

lpwcx: Points to a WNDCLASSEX structure. You must fill the structure with
the appropriate class attributes before passing it to the function.
```

The only parameter is a pointer to a structure. First some basics about structures:

A structure is a collection of variables (data). It is defined with STRUCT:

```
SOMESTRUCTURE STRUCT
dword1 dd ?
dword2 dd ?
some_word dw ?
abyte db ?
anotherbyte db ?
SOMESTRUCTURE ENDS
```

(the structure name doesn't have to be in capitals)

You can declare your variables just like in the uninitialized data section, with question marks. Now you can create a structure from the definition:

Initialized

```
Initializedstructure SOMESTRUCTURE <100,200,10,'A',90h>
```

Uninitialized

```
UnInitializedstructure SOMESTRUCTURE <>
```

In the first example, a new structure is created (with `Initializedstructure` holding its offset), and each data element of the structure is filled with an initial value. The second example just tells masm to reserve memory for the structure, and each data element is set to 0 initially. After creating a structure you can access it in your code:

```
mov eax, Initializedstructure.some_word
; eax will hold 10 now
inc UnInitializedstructure.dword1
; the dword1 of the structure is increased by one
```

This is how this structure would be stored in memory

Memory location	Contents
offset of Initializedstructure	100 (dword, 4 bytes)
offset of Initializedstructure + 4	200 (dword, 4 bytes)
offset of Initializedstructure + 8	10 (word, 2 bytes)
offset of Initializedstructure + 1	065 or 'A' (1 byte)
offset of Initializedstructure + 1	190h (1 byte)

12.3 - WNDCLASSEX

Enough about structures for now, let's proceed with RegisterClassEx. In the win32 programmer's reference you can look up the defintion of the WNDCLASSEX structure.

```
typedef struct _WNDCLASSEX { // wc
UINT cbSize;
UINT style;
WNDPROC lpfnWndProc;
int cbClsExtra;
int cbWndExtra;
HANDLE hInstance;
HICON hIcon;
HCURSOR hCursor;
HBRUSH hbrBackground;
LPCTSTR lpszMenuName;
LPCTSTR lpszClassName;
HICON hIconSm;
} WNDCLASSEX;
```

Explanation:

cbSize	Size of the WNDCLASSEX structure. Used for validation by windows. You can get this size using SIZEOF: mov ws.cbSize, SIZEOF WNDCLASSEX
style	Specifies the styles for the class (redraw flags, if the window should have a scrollbar etc.)
lpfnWndProc	Pointer to a window procedure (more on this later)
cbClsExtra	Number of extra bytes to allocate following the window class structure. Not important to us.
cbWndExtra	Number of extra bytes to allocate following the window instance. Also not important to us.
hInstance	Instance handle of your program. You can get this handle with the GetModuleHandle function.
hIcon	Handle of an icon resource for the window.
hCursor	Handle of a cursor resource for the window.
hbrBackground	Handle to a brush for painting the background, or one of the standard brush types like COLOR_WINDOW, COLOR_BTNFACE , COLOR_BACKGROUND.
lpszMenuName	Pointer to a null-terminated string that specifies resource name of the class menu. This can also be a resource ID.

lpszClassName	Points to a null-terminated string that specifies the class name for the window.
hIconSm	Handle to a small icon that is associated with the window class.

*Create a new folder in your win32 folder called **firstwindow** and create a new file **window.asm** in this folder with the following contents:*

```
.486
.model flat, stdcall
option casemap:none

includelib \masm32\lib\kernel32.lib
includelib \masm32\lib\user32.lib
includelib \masm32\lib\gdi32.lib
include \masm32\include\windows.inc
include \masm32\include\kernel32.inc
include \masm32\include\user32.inc
include \masm32\include\gdi32.inc
```

*Then create a .bat file called **make.bat**. Paste this text into it:*

```
@echo off
ml /c /coff window.asm
link /subsystem:windows window.obj
pause>nul
```

<view code>

From now on, to save space, only pieces of the full code are displayed, you can click on <view code> to display the full code at that point in the tutorial. The full code is displayed in a new window.

12.4 - Registering a class

Now we will register the class in a procedure called WinMain. This procedure contains the window initialization.

add this to your assembly file:

```
WinMain PROTO STDCALL :DWORD, :DWORD, :DWORD

.data?

hInstance dd ?

.code

invoke GetModuleHandle, NULL
mov hInstance, eax
invoke WinMain, hInstance, NULL, NULL, SW_SHOWNORMAL

end start
```

This code will get the handle of the module with *getmodulehandle*, put the handle in the hInstance variable. This module handle is used very often in the windows API. Then it calls the procedure WinMain. This is not an API function, but a procedure we will define now. The prototype is : WinMain PROTO STDCALL :DWORD, :DWORD, :DWORD, :DWORD, so a function with 4 parameters.:

<view code>

*Now put this code before the **end start**:*

```
WinMain proc hInst:DWORD, hPrevInst:DWORD, CmdLine:DWORD, CmdShow:DWORD

ret
WinMain endp
```

You don't have to use this winmain procedure at all, but it's a very common way of initializing your program. Visual C initializes the parameters of this functions automatically, but we have to do it ourselves. Don't care about hPrevInst and CmdLine for now, focus on hInst and CmdShow. hInst is the Instance handle (= module handle), CmdShow is a flag that defines how the window should be shown. (More on this can be found in the API reference about ShowWindow).

The "invoke WinMain, hInstance, NULL, NULL, SW_SHOWNORMAL" in the previous code calls this function with the right hInstance and show flag. Now we can write our initialization code in WinMain:

```
WinMain proc hInst:DWORD, hPrevInst:DWORD, CmdLine:DWORD, CmdShow:DWORD
LOCAL wc:WNDCLASSEX
LOCAL hwnd:DWORD

ret
WinMain endp
```

These are the two local variables we will need in this procedure.

```
.data

ClassName db "FirstWindowClass",0

.code

WinMain proc hInst:DWORD, hPrevInst:DWORD, CmdLine:DWORD, CmdShow:DWORD
LOCAL wc:WNDCLASSEX
LOCAL hwnd:DWORD
; now set all the structure members of the WNDCLASSEX structure wc:
mov  wc.cbSize,SIZEOF WNDCLASSEX
mov  wc.style, CS_HREDRAW or CS_VREDRAW
mov  wc.lpfnWndProc, OFFSET WndProc
mov  wc.cbClsExtra,NULL
mov  wc.cbWndExtra,NULL
push hInst
pop  wc.hInstance
mov  wc.hbrBackground,COLOR_WINDOW
mov  wc.lpszMenuName,NULL
mov  wc.lpszClassName,OFFSET ClassName
invoke LoadIcon,NULL,IDI_APPLICATION
mov  wc.hIcon, eax
mov  wc.hIconSm, eax
invoke LoadCursor,NULL,IDC_ARROW
mov  wc.hCursor,eax
invoke RegisterClassEx, ADDR wc

ret
WinMain endp
```

Let's see what happens:

```
mov  wc.cbSize,SIZEOF WNDCLASSEX
mov  wc.style, CS_HREDRAW or CS_VREDRAW
mov  wc.lpfnWndProc, OFFSET WndProc
mov  wc.cbClsExtra,NULL
mov  wc.cbWndExtra,NULL
```

The size of the structure is initialized (this is required by RegisterClassEx). The style of the class is set

to "CS_HREDRAW or CS_VREDRAW", then the offset of the windows procedure is set. You will find out what the window procedure is later, for now remember that you need the address of the WndProc procedure, which you can get with 'offset WndProc'. cbClsExtra and cbWndExtra are not used by us so set them to NULL.

```
push hInst
pop  wc.hInstance
```

The wc.hInstance is set to the hInst parameter of WinMain. Why don't we use: `mov wc.hInstance, hInst`? Because the mov instruction does not allow to move from one memory location to another. With the push/pop method, the value to move is pushed on stack, and than popped of into the destination.

```
mov  wc.hbrBackground, COLOR_WINDOW
mov  wc.lpszMenuName, NULL
mov  wc.lpszClassName, OFFSET ClassName
```

The background color of the class is set to COLOR_WINDOW, no menu is defined (NULL), and the lpszClassName is set to a pointer to the null-terminated classname-string: "FirstWindowClass". This should be a unique name defined for your own application.

```
invoke LoadIcon, NULL, IDI_APPLICATION
mov  wc.hIcon, eax
mov  wc.hIconSm, eax
```

The window needs an icon, but because we need a **handle** to an icon, we use LoadIcon to load an icon and get a handle. LoadIcon has two parameters: hInstance, and lpIconName. hInstance is the module handle whose executable file contains the icon. lpIconName is a pointer to a string that is the name of the icon resource or a resource ID. If you use NULL as hInstance, you can choose from some standard icons. (Which we do because we don't have an icon resource here). hIconSm is the small icon, you can use the same handle for it.

```
invoke LoadCursor,NULL,IDC_ARROW
mov  wc.hCursor,eax
```

Same thing for the cursor, just NULL as hInstance, and a standard cursor type: IDC_ARROW, the standard windows arrow.

```
invoke RegisterClassEx, ADDR wc
```

Now finally register the class using RegisterClassEx with a pointer to the WNDCLASSEX structure wc as parameter.

<view code>

12.5 - Creating the window

Now you've registered a class you can create a window from it:

```
HWND CreateWindowEx(

DWORD dwExStyle, // extended window style
LPCTSTR lpClassName, // pointer to registered class name
LPCTSTR lpWindowName, // pointer to window name
DWORD dwStyle, // window style
int x, // horizontal position of window
int y, // vertical position of window
int nWidth, // window width
int nHeight, // window height
HWND hWndParent, // handle to parent or owner window
HMENU hMenu, // handle to menu, or child-window identifier
HINSTANCE hInstance, // handle to application instance
LPVOID lpParam // pointer to window-creation data
```

```
);
```

dwExStyle and dwStyle are two parameters which determine the style of the window.
lpClassName is a pointer to your registered classname.
lpWindowName is the name of your window (this will be in the caption of your window if it has one)
x, y, nWidth, nHeight determine the position and size of your window.
hWndParent is the handle of the window that owns the new window. Can be null if you don't have a parent window.
hMenu is a handle of a windows menu (discussed in a later tutorial, null for now)
hInstance is the application instance handle
lpParam is an extra value you can use in your program

```
.data

AppName "FirstWindow",0

.code

INVOKE CreateWindowEx,NULL,ADDR ClassName,ADDR AppName,\
WS_OVERLAPPEDWINDOW,CW_USEDEFAULT,\
CW_USEDEFAULT,400,300,NULL,NULL,\
hInst,NULL
mov hwnd, eax
invoke ShowWindow, hwnd, SW_SHOWNORMAL
invoke UpdateWindow, hwnd
```

(note that the \-character makes the assembler read to the next line as if it where on the same line.)

Our code will create a new windows, with our classname we've just registered. The title will be "FirstWindow" (AppName), the style is WS_OVERLAPPEDWINDOW, which is a style that creates an overlapped window with a caption, system menu, resizable border and a minimize/maximize-box. CW_USEDEFAULT as x and y position will make windows use the default position for the new window. The (initial) size of the window is 400x300 pixels.

The return value of the function is the window handle, HWND, which is stored in the *local* variable hwnd. Then the window is showed with ShowWindow. UpdateWindow ensures that the window will be drawn.

12.6 - Message Loop

A window can communicate with your program and other windows using messages. The window procedure (see later on in this tutorial) is called whenever a message is pending for a specific window. Each window has a message loop or message pump. This is an endless loop that checks if there's a message for your window, and if it is, passes the message to the dispatchmessage function. This function will call your window procedure. The message loop and the windows procedure are two different things!!!

```
WinMain proc
hInst:HINSTANCE,hPrevInst:HINSTANCE,CmdLine:LPSTR,CmdShow:DWORD
LOCAL wc:WNDCLASSEX
LOCAL hwnd:DWORD
LOCAL msg:MSG ;<<<NEW

........

.WHILE TRUE
invoke GetMessage, ADDR msg,NULL,0,0
.BREAK .IF (!eax)
invoke TranslateMessage, ADDR msg
invoke DispatchMessage, ADDR msg
.ENDW
```

This is what the message loop looks like. The .WHILE TRUE, .ENDW loop goes on until eax is 0. GetMessage returns 0 if it receives the message WM_QUIT, which should close the window so the program must exit the messageloop whenever GetMessage returns 0. If it doesn't, the message is passed to TranslateMessage (this function translates keypresses to messages) and then the message is dispatched by windows using the DispatchMessage function. The message itself in a message loop consists of a MSG structure (LOCAL msg:MSG is added to the procedure, which adds a local message structure called msg). You can use this message loop in all your programs.

12.7 - Window Procedure

Messages will be send to the window procedure. A window procedure should always look like this:

```
WndProc PROTO STDCALL :DWORD, :DWORD, :DWORD, :DWORD

.code

WndProc proc hWnd:DWORD, uMsg:DWORD, wParam:DWORD, lParam:DWORD
mov eax, uMsg
.IF eax==XXXX
.ELSEIF eax==XXXX
.ELSE
    invoke DefWindowProc, hWnd, uMsg, wParam, lParam
.ENDIF
ret
WndProc endp
```

The window procedure should always have 4 parameters:

hWnd contains the window handle.
uMsg is the message
wParam is the first parameter for the message (message specific)
lParam is the second parameter for the message (message specific)

Messages that the window does not process should be passed to DefWindowProc, which takes care of the processing. An example window procedure:

```
WndProc proc hWnd:DWORD, uMsg:DWORD, wParam:DWORD, lParam:DWORD
mov eax, uMsg
.IF eax==WM_CREATE
    invoke MessageBox, NULL, ADDR AppName, ADDR AppName, NULL
.ELSEIF eax==WM_DESTROY
    invoke PostQuitMessage, NULL
.ELSE
    invoke DefWindowProc, hWnd, uMsg, wParam, lParam
.ENDIF
ret
WndProc endp
```

This code displays the name of the application when the window is initialized. Also note that I've added the processing of the WM_DESTROY message. This message is sent if the window should close. The application should react to it with a PostQuitMessage.

Now take a look at the final code:

<view code>

[top]

14.0 - N/A

Sorry, this tutorial is not available yet. This tutorial is still under construction. Please check again later!

[top]

Tools

This section describes the use of various tools. Please select a tool:

Assembler	• Masm
Linker	• Link
Resource compiler	• brc32 • rc

[top]

Masm

Masm is the assembler I use. Others are tasm and nasm but I prefer masm myself (see why). The use of tasm and nasm (and other assemblers) is not discussed here.

Usage

Masm is **ml.exe**. The version I use is "Macro Assembler Version 6.14.8444". Syntax:

```
ML [ /options ] filelist [ /link linkoptions ]
```

Here are some important options:

/c	**assemble without linking** You will use this option most of the time as you will be using an external linker like link.exe to link your files.
/coff	**generate COFF format object file** This is the generic file format for the microsoft linker.
/Fo<file>	**name object file** Can be used if you want to output an object file with an other name than your source file
/G<c|d|z>	**Use Pascal, C, or Stdcall calls** Select the default calling convention for your procedures.
/Zi	**Add symbolic debug info** Use this option if you want to use a debugger.
/I<name>	**Set include path** Defines your default include path

[top]

Link

Link is the standard linker of microsoft.

Usage

Link is **link.exe**. The version I use is "Incremental Linker Version 5.12.8078". Syntax:

```
LINK [options] [files] [@commandfile]
```

Here are some important options:

/DEBUG	**Debug** This will create debug information. Use this option when you want to use a debugger.
/DEBUGTYPE:CV|COFF	**Debugtype: codeview / coff** Selects the output format of the debug info. This depends on your debugger. Softice and the visual c++ debugger both can handle CV (codeview)
/DEF:filename	**DEF file** Specifies the definition file (.def). Used with dll's to export functions.
/DLL	**DLL** Outputs a dynamic link library instead of an executable.
/LIBPATH:path	**Libpath** Specifies the path of your library files (*.lib).
/I<name>	**Set include path** Defines your default include path.
/OUT:filename	**Out:filename** Can override the default output filename.
/SUBSYSTEM:{...}	**Subsystem** Selects the OS the program should run on: NATIVE|WINDOWS|CONSOLE|WINDOWSCE|POSIX

[top]

BRC32

This is a resource compiler for borland resource files. If you create your resources with borland resource workshop, it is best tot use this compiler as it is fully compatible with brw.

Usage

The version I use is 5.00. Syntax:

```
BRC32 [options ...] filename
```

Here are some important options:

-r	**Compile only. Do not bind resources** If this option is NOT set, brc32 will link the compiled resource file to your executable. If it is set, it will output a .res file (compiled resource), which you can manually link with link.exe. I always use this option.
-fo*filename*	**Set output res filename** Output file for .res file.
-fe*filename*	**Set output exe filename** Output file for .exe file.

Remarks

I always use brc32 for my resource files because I create them with borland resource workshop. I prefer using link.exe to link the resource to the executable, so my compiling process is as follows:

brc32 -r resourcefile.rc
link *..otheroptions...* myprogram.obj **resourcefile.res**

[top]

RC

RC.exe is the resource compiler from microsoft. I don't use it myself, but here is some information about it.

Usage

The version I use is 5.00.1823.1 - Build 1823. Syntax:

```
rc [options] .RC input file
```

Here are some important options:

-fo	**Rename .res file** Output file for .res file.

[top]

Atop CHM to PDF Converter

Purchase Atop CHM to PDF Converter

Atop CHM to PDF Converter is distributed as "Shareware". This means the software is try before you buy software, the trial version includes some limitation, if you would like to use it in full version, you have to register your copy.

Online Order

Please Visit http://www.chmconverter.com/order.htm to order online.

Price for Registration Atop CHM to PDF Converter via the Web:
Single User License Unit Price:(US$29.95)

If you have any question or meet any problem on ordering, please contact us via support@chmconverter.com . We will try to help you as quickly as possible.

Note: For more information about upgrading to take full advantage of all the features describe, please visit www.chmconverter.com.
