December 10, 2003

JEFF HUANG (huang6@uiuc.edu)

Windows Assembly Programming Tutorial

Version 1.02

Table of Contents

"This is for all you folks out there, who want to learn the magic art of Assembly programming."
- MAD

Introduction

I have just started learning Windows assembly programming yesterday, and this tutorial is being written while I'm learning the language. I am learning assembly from reading various tutorials online, reading books, and ask questions in newsgroups and IRC. There are a lot of assembly programming tutorials online, but this tutorial will focus on Windows programming in x86 assembly. Knowledge of higher level programming languages and basic knowledge of computer architecture is assumed.

Why Assembly?

Assembly has several features that make it a good choice many some situations.

1. **It's fast** – Assembly programs are generally faster than programs created in higher level languages. Often, programmers write speed-essential functions in assembly.

2. **It's powerful** – You are given unlimited power over your assembly programs. Sometimes, higher level languages have restrictions that make implementing certain things difficult.

3. **It's small** – Assembly programs are often much smaller than programs written in other languages. This can be very useful if space is an issue.

Why Windows?

Assembly language programs can be written for any operating system and CPU model. Most people at this point are using Windows on x86 CPUs, so we will start off with programs that run in this environment. Once a basic grasp of the assembly language is obtained, it should be easy to write programs for different environments.

Chapter 1

I. Getting Started

To program in assembly, you will need some software, namely an assembler and an editor. There is quite a good selection of Windows programs out there that can do these jobs.

Assemblers

An assembler takes the written assembly code and converts it into machine code. Often, it will come with a linker that links the assembled files and produces an executable from it. Windows executables have the .exe extension. Here are some of the popular ones:

Note:
There will be several directives and macros used in this tutorial that are only available in MASM, so it's highly encouraged that you start with this first

1. **MASM** – This is the assembler this tutorial is geared towards, and you should use this while going through this tutorial. Originally by Microsoft, it's now included in the MASM32v8 package, which includes other tools as well. You can get it from http://www.masm32.com/.

2. **TASM** – Another popular assembler. Made by Borland but is still a commercial product, so you can not get it for free.

3. **NASM** – A free, open source assembler, which is also available for other platforms. It is available at http://sourceforge.net/projects/nasm/. Note that NASM can't assemble most MASM programs and vice versa.

Editors

An editor is where you write your code before it is assembled. Editors are personal preferences; there are a LOT of editors around, so try them and pick the one you like.

1. **Notepad** – Comes with Windows; although it lacks many features, it's quick and simple to use.

2. **Visual Studio** – Although it's not a free editor, it has excellent syntax highlighting features to make your code much more readable.

3. **Other** – There are so many Windows editors around that it would be pointless to name all of them. Some of the more popular ones are:

    a. Ultraedit (my personal favorite) http://www.ultraedit.com/
    b. Textpad http://www.textpad.com/
    c. VIM http://www.vim.org/
    d. Emacs http://www.gnu.org/software/emacs/emacs.html
    e. jEdit http://www.jedit.org/

Chapter 2

II. Your First Program

Now that we have our tools, let's begin programming! Open up your text editor and following the instructions below. This is the most commonly written program in the world, the "Hello World!" program.

Console Version

The console version is run from the Windows console (also known as the command line). To create this program, first paste the following code into your text editor and save the file as "hello.asm".

```
.386
.model flat, stdcall
option casemap :none

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

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

.data
       HelloWorld db "Hello World!", 0

.code
start:
       invoke StdOut, addr HelloWorld
       invoke ExitProcess, 0

end start
```

Now, open up the command line by going into the Start Menu, clicking on the Run… menu item, and typing in "`cmd`" without the quotes. Navigate to the directory "hello.asm" is saved in, and type "`\masm32\bin\ml /c /Zd /coff hello.asm`". Hopefully, there are no errors and your program has been assembled correctly! Then we need to link it, so type "`\masm32\bin\Link /SUBSYSTEM:CONSOLE hello.obj`". Congratulations! You have successfully created your first assembly program. There should be a file in the folder called Hello.exe. Type "`hello`" from the command line to run your program. It should output "Hello World!".

So that was quite a bit of code needed to just display Hello World! What does all that stuff do? Let's go through it line by line.

```
.386
```

This is the assembler directive which tells the assembler to use the 386 instruction set. There are hardly any processors out there that are older than the 386 nowadays. Alternatively, you can use `.486` or `.586`, but `.386` will be the most compatible instruction set.

```
        .model flat, stdcall
```
`.MODEL` is an assembler directive that specifies the memory model of your program. `flat` is the model for Windows programs, which is convenient because there is no longer a distinction between 'far' and 'near' pointers. `stdcall` is the parameter passing method used by Windows functions, which means you need to push your parameters from right-to-left.

```
        option casemap :none
```
Forces your labels to be case sensitive, which means `Hello` and `hello` are treated differently. Most high level programming languages are also case sensitive, so this is a good habit to learn.

```
        include \masm32\include\windows.inc
        include \masm32\include\kernel32.inc
        include \masm32\include\masm32.inc
```
Include files required for Windows programs. `windows.inc` is always included, since it contains the declarations for the Win32 API constants and definitions. `kernel32.inc` contains the ExitProcess function we use; `masm32.inc` contains the `StdOut` function, which although is not a built in Win32 function, is added in MASM32v8.

```
        includelib \masm32\lib\kernel32.lib
        includelib \masm32\lib\masm32.lib
```
Functions need libraries in order to function (no pun intended), so these libraries are included for that purpose.

```
        .data
```
All initialized data in your program follow this directive. There are other directives such as `.data?` and `.const` that precede uninitialized data and constants respectively. We don't need to use those in our Hello World! program though.

```
        HelloWorld db "Hello World!", 0
```
`db` stands for 'define byte' and defines `HelloWorld` to be the string "Hello World!" followed by a `NUL` character, since ANSI strings have to end in `NULL`.

```
        .code
```
This is the starting point for the program code.

```
        start:
```
All your code must be after this label, but before `end start`.

```
        invoke StdOut, addr HelloWorld
```
`invoke` calls a function and the parameter, `addr HelloWorld` follows it. What this line does is call `StdOut`, passing in `addr HelloWorld`, the address of "Hello World!". Note that `StdOut` is a function that's only available in MASM32 and is simply a macro that calls another function to output text. For other assemblers, you will need to use write more code and use the win32 function, `WriteConsole`.

```
        invoke ExitProcess, 0
```
This should be fairly obvious. It passes in `0` to the `ExitProcess` function, exiting the process.

Windows Version

We can also make a Windows version of the Hello World! program. Paste this text into your text editor and save the file as "hellow.asm".

```
.386
.model flat, stdcall
option casemap :none

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

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

.data
       HelloWorld db "Hello World!", 0

.code
start:
       invoke MessageBox, NULL, addr HelloWorld, addr HelloWorld, MB_OK
       invoke ExitProcess, 0

end start
```

Now, open up the command line again and navigate to the directory "hellow.asm" is saved in. Type "`\masm32\bin\ml /c /Zd /coff hellow.asm`", then "`\masm32\bin\Link /SUBSYSTEM:WINDOWS hellow.obj`". Note that the subsystem is `WINDOWS` instead of `CONSOLE`. This program should pop up a message box showing "Hello World!".

There only 3 lines of code that are different between the Windows and Console version. The first 2 have to do with changing the `masm32` include and library files to `user32` include and library files since we're using the `MessageBox` function instead of `StdOut` now. The 3rd change is to replace the `StdOut` function with the MessageBox function. That's all!

ADDR vs OFFSET

In our Hello World! examples, we used 'addr' to get the address of the string "Hello World!". There is also another similar directive, 'offset', although the purpose of both is to get the memory address of variables during execution. The main difference is that 'offset' can only get the address of global variables, while addr can get the address of both global variables and local variables. We haven't discussed local variables yet, so don't worry about it. Just keep this in mind.

III. Basic Assembly

So now we are able to get a simple program up and running. Let's move to the core of the tutorial – basic assembly syntax. These are the fundamentals you need to know in order to write your own assembly programs.

CPU Registers

Registers are special memory locations on the CPU. At this point, we'll assume the reader is programming for computers using 386 or later processors. Older processors are very rare at this time, so it would be a waste of time to learn about them. One important difference between older and later processors is that the pre-386 processors are 16-bit instead of 32-bit.

Note:
Although they are called general purpose registers, only the ones marked with a * should be used in Windows programming

There are 8 32-bit general purpose registers. The first 4, eax, ebx, ecx, and edx can also be accessed using 16 or 8-bit names. ax gets the first 16 bits of eax, al gets the first 8 bits, and ah gets bits 9-16. The other registers can be accessed in a similar fashion. Supposedly, these registers can be used for anything, although most have a special use:

Address	Name	Description
EAX*	Accumulator Register	calculations for operations and results data
EBX	Base Register	pointer to data in the DS segment
ECX*	Count Register	counter for string and loop operations
EDX*	Data Register	input/output pointer
ESI	Source Index	source pointer for string operations
EDI	Destination Index	destination pointer for string operations
ESP	Stack Pointer	stack pointer, s**hould not be used**
EBP	Base Pointer	pointer to data on the stack

There are 6 16-bit segment registers. They define segments in memory:

Address	Name	Description
CS	Code Segment	where instructions being executed are stored
DS, ES, FS, GS	Data Segment	data segment
SS	Stack Segment	where the stack for the current program is stored

Lastly, there are 2 32-bit registers that don't fit into any category:

Address	Name	Description
EFLAGS	Code Segment	status, control, and system flags
EIP	Instruction Pointer	offset for the next instruction to be executed

Basic Instruction Set

The x86 instruction set is extremely huge, but we usually don't need to use them all. Here are some simple instructions you should know to get you started:

Instruction	Description
ADD* reg/memory, reg/memory/constant	Adds the two operands and stores the result into the first operand. If there is a result with carry, it will be set in CF.
SUB* reg/memory, reg/memory/constant	Subtracts the second operand from the first and stores the result in the first operand.
AND* reg/memory, reg/memory/constant	Performs the bitwise logical AND operation on the operands and stores the result in the first operand.
OR* reg/memory, reg/memory/constant	Performs the bitwise logical OR operation on the operands and stores the result in the first operand.
XOR* reg/memory, reg/memory/constant	Performs the bitwise logical XOR operation on the operands and stores the result in the first operand. Note that you can not XOR two memory operands.
MUL reg/memory	Multiplies the operand with the Accumulator Register and stores the result in the Accumulator Register.
DIV reg/memory	Divides the Accumulator Register by the operand and stores the result in the Accumulator Register.
INC reg/memory	Increases the value of the operand by 1 and stores the result in the operand.
DEC reg/memory	Decreases the value of the operand by 1 and stores the result in the operand.
NEG reg/memory	Negates the operand and stores the result in the operand.
NOT reg/memory	Performs the bitwise logical NOT operation on the operand and stores the result in the operand.
PUSH reg/memory/constant	Pushes the value of the operand on to the top of the stack.
POP reg/memory	Pops the value of the top item of the stack in to the operand.
MOV* reg/memory, reg/memory/constant	Stores the second operand's value in the first operand.
CMP* reg/memory, reg/memory/constant	Subtracts the second operand from the first operand and sets the respective flags. Usually used in conjunction with a JMP, REP, etc.
JMP** label	Jumps to label.
LEA reg, memory	Takes the offset part of the address of the second operand and stores the result in the first operand.
CALL subroutine	Calls another procedure and leaves control to it until it returns.
RET	Returns to the caller.
INT constant	Calls the interrupt specified by the operand.

* Instructions can not have memory as both operands

** This instruction can be used in conjunction with conditions. For example, JNB (not below) jumps only when CF = 0.

The latest complete instruction set reference can be obtained at: http://www.intel.com/design/pentium4/manuals/index.htm.

Push and Pop

Push and pop are operations that manipulate the stack. Push takes a value and adds it on top of the stack. Pop takes the value at the top of the stack, removes it, and stores it in the operand. Thus, the stack uses a last in first out (LIFO) approach. Stacks are common data structures in computers, so I recommend you learn about them if you are not comfortable with working with stacks.

Invoke

The Invoke function is specific to MASM, and can be used to call functions without having to push the parameters beforehand. This saves us a lot of typing.

For example:

```
      invoke SendMessage, [hWnd], WM_CLOSE, 0, 0
```

Becomes:

```
      push 0
      push 0
      push WM_CLOSE
      push [hWnd]
      call [SendMessage]
```

Example Program

Here is a fully function program that shows how to use some of the instructions and registers. See if you can figure it out.

```
.386
.model flat, stdcall
option casemap :none
include \masm32\include\windows.inc
include \masm32\include\kernel32.inc
include \masm32\include\masm32.inc
includelib \masm32\lib\kernel32.lib
includelib \masm32\lib\masm32.lib

.data
    ProgramText db      "Hello World!", 0
    BadText     db      "Error: Sum is incorrect value", 0
    GoodText    db      "Excellent! Sum is 6", 0
    Sum         sdword  0

.code
start:
                     ;                                 eax
        mov ecx, 6   ; set the counter to 6            ?
        xor eax, eax ; set eax to 0                    0
_label: add eax, ecx ; add the numbers                 ?
        dec ecx      ;     from 0 to 6                 ?
        jnz _label   ;                                 21
        mov edx, 7   ;                                 21
        mul edx      ; multiply by 7                   147
        push eax     ; pushes eax into the stack
        pop Sum      ; pops eax and places it in Sum
        cmp Sum, 147 ; compares Sum to 147
        jz _good     ; if they are equal, go to _good
_bad:   invoke StdOut, addr BadText
        jmp _quit
_good:  invoke StdOut, addr GoodText
_quit:  invoke ExitProcess, 0

end start
```

Note:
The ';' character denotes comments. Anything following that character does not get assembled. It's a good idea to put hints and notes in comments to make your code easier to read.

IV. Basic Windows

Windows programs are usually composed of one or more windows. Thus, to be a real Windows programmer, one must at least know how to make a simple window. Unfortunately, it's not that easy, but this will guide you through it.

Preliminaries

There are a few more topics in assembly we should discuss before diving into Windows programming. Let's take a moment to go over these prerequisites.

Macros

MASM has a handful of macros that make assembly programming much easier. We've already seen 'invoke', which simplifies function calls. Here are a few others; their usage should be obvious in you've programmed in a high level language before:

- `.if, .else, .endif`
- `.while, .break, .endw`

Functions

Similarly to high level languages, MASM let's you define functions to make your code much easier to read. Their syntax looks like this:

```
<name> proc <var1>:<var1 type>, <var2>:<var2 type>, ...
      <function code>
      ret
<name> endp
```

The return value is stored in the eax register, and so the function is called using:

```
invoke <name>, param1, param2, ...
```

And the return can be obtained using the instruction:

```
mov RetVal, eax
```

Variables

Variables are allocated in the memory and let you store data. They can be very useful if you don't have enough registers. There are two types of variables – global variables and local variables. Global variables are placed in the .data section if they are initialized, the .data? section if they are uninitialized, or in the .const section if they are initialized and won't be changed. The syntax to declare global variables is:

```
<name> <type> <value, or ? if uninitialized>
```

Local variables are placed inside a function, and are temporary storage for use inside the function. They can not be initialized when created. Their syntax is:

```
local <name>:<type>
```

There are several variable types you will come across. Some good ones to know are 'byte', 'word' (4 bytes), and 'dword' (8 bytes). There are more, but they are usually just the same as one of these three types but with a different name

A Simple Window

Windows programs have two main parts –WinMain creates the window and contains something called the message loop. The message loop watches for messages and dispatches them. The second part is the callback function, WndProc, which is where the messages are sent to. This is where you handle your mouse events, repainting, etc.

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

Our usual necessities.

```
WinMain proto :DWORD, :DWORD, :DWORD, :DWORD
```

This is a function prototype. It let's us call the WinMain function later in the program. It can be compared to a C/C++ function declaration.

```
.data
        ClassName   db "WinClass", 0
        AppName     db "Simple Window", 0
```

We declare our string variables.

```
.data?
        hInstance   HINSTANCE ?
```

hInstance stores the handle to the instance of the module to be associated with the window. We will need to pass it into the CreateWindow function later.

Note:
From this point, we'll assume you can look up Windows functions in the MSDN Library. It has information on function parameters, return values, and anything else you may need to know. Take a look at the Additional Resources section for more information about the MSDN Library.

```
.code
start:
        invoke GetModuleHandle, NULL
        mov hInstance, eax
        invoke WinMain, hInstance, NULL, NULL, 0
        invoke ExitProcess, eax
```

Gets the module handle and stores it into hInstance. Then it calls WinMain and exits. WinMain is the core of our program, so we will look into it more.

```
        WinMain proc hInst:HINSTANCE, hPrevInst:HINSTANCE, CmdLine:LPSTR,
CmdShow:DWORD
                local wc:WNDCLASSEX
                local msg:MSG
                local hwnd:HWND
```

This is the beginning of our WinMain function. We declare three local variables: wc, msg, and hwnd. wc stores the window class we make; a window class is a template for creating windows. msg stores the messages that the message loop retrieves. hwnd stores the handle to the window.

Note:
In Windows, the 'or' operator is often used to combine flags in parameters.

```
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   hInstance
pop    wc.hInstance
mov    wc.hbrBackground, COLOR_WINDOW+1
```

```
        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
```

All this does is fill in the wc struct we declared earlier on. RegisterClassEx is then called, taking in wc as the parameter. For more information and each member is wc, take a look at the WNDCLASSEX structure in the MSDN Library.

```
        invoke CreateWindowEx, 0, addr ClassName, addr AppName, WS_OVERLAPPEDWINDOW
or WS_VISIBLE, CW_USEDEFAULT, CW_USEDEFAULT, CW_USEDEFAULT, CW_USEDEFAULT, NULL,
NULL, hInst, NULL
        mov   hwnd, eax
```

Calls the CreateWindowEx function that actually creates the window. Many parameters are passed in to specify how to make the window. The handle of the window is returned and stored in hwnd.

```
    .while TRUE
        invoke GetMessage, addr msg, NULL, 0, 0
        .break .if (!eax)
        invoke TranslateMessage, addr msg
        invoke DispatchMessage, addr msg
    .endw
```

This is a while loop which is the message loop mentioned earlier. When an input event occurs, Windows translates the event into a message and passes the message into the program's message queue. GetMessage retrieves these messages and stores them in msg. Then, TranslateMessage changes key messages into character messages. Finally, DispatchMessage sends the message to WndProc where it is processed.

```
                mov eax, msg.wParam
                ret
        WinMain endp
```

The return value is stored into msg.wParam and WinMain is ended.

```
        WndProc proc hWnd:HWND, uMsg:UINT, wParam:WPARAM, lParam:LPARAM
                .if uMsg == WM_DESTROY
                        invoke PostQuitMessage, 0
                .else
                        invoke DefWindowProc, hWnd, uMsg, wParam, lParam
                        ret
                .endif
                xor eax, eax
                ret
        WndProc endp
```

The WndProc function is where messages are processed. The only message that must be processed is WM_DESTROY, which calls PostQuitMessage to quit. If there are other events you want processed, you would add them here. Common messages to process are WM_CREATE (when the window is created), WM_PAINT (when the window needs repainting), and WM_CLOSE (when the window is closed). Anything that isn't handled is passed on to the DefWindowProc function which is the default handler.

```
        end start
```

That's all! You now know how to create a window!

V. More Assembly and Windows

Here are some more resources to expand your knowledge of assembly and Windows programming: string manipulation, working with files, and controls for your Windows forms.

String Manipulation

Strings, arrays of characters, are an essential part to any program. They are usually helpful if you want to display text or ask for input from the user. They use the following registers: esi, edi, ecx, eax, eflag's direction flag. The direction flag is to specify which direction to move along the string. Some common string instructions are movsb, cmpsb, stasb, and stosb. To manipulate strings, you use some form of rep? on a string instruction. Here is a table of which rep? prefix to use with the string instructions:

prefix	**string instruction**	**description**
rep	movsb	copies a string
repe	cmpsb	compares a string
repne	scasb	scans a string for a character
rep	stosb	sets a character in a string

Here's an example of how to copy a string:

```
cld                             ; sets the direction flag to forward
mov esi, source                 ; move the source address in to esi
mov edi, dest                   ; move the destination address in to edi
mov ecx, length                 ; move the length to copy in to ecx
rep movsb                       ; copy length bytes from esi to edi
```

File Management

In the old DOS world, files would be manipulated using interrupts. In Windows, we use Windows functions in order to access files. These are the 4 functions we can use:

CreateFile – Creates or opens a file, and returns its handle.

ReadFile – Reads data from a file.

WriteFile – You guessed it! Writes data to a file.

CloseHandle – Closes the handle that you obtained using CreateFile.

Memory

In order to read the contents of a file, you will need to allocate some memory to store the data. Memory has to be allocated, locked, used however you want, unlocked, and freed. The functions that do this are GlobalAlloc, GlobalLock, GlobalUnlock, and GlobalFree. Pretty easy, huh?

Example Program

This program reads the contents of "c:\test.txt" and outputs it to a MessageBox.

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

The usual suspects.

```
.data
        FileName db "c:\test.txt", 0
.data?
        hFile       HANDLE  ?
        hMemory     HANDLE  ?
        pMemory     DWORD   ?
        ReadSize    DWORD   ?
```

We define our string and declare 4 variables to be used later on.

```
.const
        MEMORYSIZE  equ     65535
```

This is how much memory we'll allocate, so we will have a lot of space to store our file.

```
.code
start:
        invoke  CreateFile,  addr  FileName,  GENERIC_READ,  FILE_SHARE_READ,
NULL, OPEN_EXISTING, FILE_ATTRIBUTE_NORMAL, NULL
        mov hFile, eax
```

Call CreateFile and store the handle of the file in hFile. It's customary to place an 'h' in front of handles and 'p' in front of pointers.

```
        invoke GlobalAlloc, GMEM_MOVEABLE or GMEM_ZEROINIT, MEMORYSIZE
        mov hMemory, eax
        invoke GlobalLock, hMemory
        mov pMemory, eax
```

Allocate and lock our memory.

```
        invoke ReadFile, hFile, pMemory, MEMORYSIZE-1, addr ReadSize, NULL
        invoke MessageBox, NULL, pMemory, addr FileName, MB_OK
```

These lines read the file into pMemory and output that. Voila!

```
        invoke GlobalUnlock, pMemory
        invoke GlobalFree, hMemory
        invoke CloseHandle, hFile
        invoke ExitProcess, NULL
end start
```

Don't forget to clean up.

Controls

Once we make a window, we'll want to put some buttons and textboxes on it. Fortunately, this is easy! The syntax is very similar to creating a window, except we won't have to call RegisterClassEx because our class will be predefined for us.

To do this, edit your WndProc from Chapter 4 by reacting to the WM_CREATE message:

```
.elseif uMsg == WM_CREATE
        invoke CreateWindowEx, NULL, addr ButtonClassName, addr ButtonText, WS_CHILD
or WS_VISIBLE or BS_DEFPUSHBUTTON, 10, 50, 80, 30, hWnd, ButtonID, hInstance, NULL
        mov hButton, eax
        invoke CreateWindowEx, WS_EX_CLIENTEDGE, addr EditClassName, NULL, WS_CHILD
or WS_VISIBLE, 10, 10, 100, 20, hWnd, EditID, hInstance, NULL
        mov hEdit, eax
```

Under the .data section, you will need to add some variables. Define EditClassName as "edit" and ButtonClassName as "button". Also, you need to have EditID and ButtonID defined to be constants. It doesn't matter what they are as long as they don't have the same ID as any other control. Also, you will need uninitialized variables, hEdit and hButton, which are of type HWND. And lastly, ButtonText needs to be a string, which will be displayed on the button.

Now we also want to know when our button has been pressed. This can be done by watching the WM_COMMAND message, which is the message a button will send if clicked.

```
        .elseif uMsg == WM_COMMAND
                mov eax, wParam
                .if ax == ButtonID
                        shr eax, 16
```

wParam contains information about the message. We should check it to see if it is the button that sent the message, since we don't want to process the message of other controls yet. shr is the shift right operator, which shifts wParam 16 bits to the right. This is a useful method to get the high 16 bits of a 32-bit register so that they can be easily accessed by ax.

```
                        .if ax == BN_CLICKED
                                <code for what happens if the button is pressed>
                        .endif
                .endif
```

So now that we know the button has been clicked, we can do something about it.

If you are interested in learning more about windows, take a look at the Additional Resources section, which lists some great books and websites for Windows programming in general.

Additional Resources

WWW

http://www.xs4all.nl/~smit/ - useful x86 assembly programming tutorials.

http://win32asm.cjb.net/ - excellent set of tutorials for Windows assembly programming.

http://board.win32asmcommunity.net/ - active online forum for asking questions related to Windows assembly programming.

Books

Programming Windows, Fifth Edition by Charles Petzold is an excellent book on Windows programming. It contains sample code for many Windows programs and covers a large range of topics on Windows programming.

Intel Pentium 4 Processors Manuals, available from http://www.intel.com/design/pentium4/manuals/ is the complete reference guide for x86 assembly programming.

The Art of Assembly Programming, by Randall Hyde, is available at http://webster.cs.ucr.edu/AoA.html and is the best and most comprehensive x86 assembly language programming book I've found

MASM32

In your \masm32\HELP\ folder, there is a file called masm32.hlp which contains the MASM32 manual. It has all the macros, registers, flags, Pentium optimization information, etc. This is a very good reference to go to on things specific to MASM32.

MSDN Library

The MSDN Library usually comes with Visual Studio, can also be viewed online at http://msdn.microsoft.com/. It contains all the Windows functions, constants, and every piece of information imaginable regarding Windows.

Newsgroups

There are currently two newsgroups that deal with x86 assembly. They are **comp.lang.asm.x86** and **alt.lang.asm**. Both are fairly high in traffic and have a knowledgeable readership.

IRC

There is an IRC (internet relay chat) channel that deals with Windows assembly programming, #win32asm on EFNet [http://www.efnet.org/]