


LAB 6, PART 1

A WIN32 GUI APPLICATION

In Lab 4, you built a Windows application that performed input and output without using the Irvine32 library: it called Win32 API functions (like WriteConsole) directly. This lab will go a bit further: you'll build a Win32 application with a message box and a main window. Moreover, you will declare the Win32 API constants and function prototypes explicitly.

Directions. This lab will be continued on Wednesday. Note your answers to the questions marked with a  symbol; you will submit these answers with your completed code on Wednesday.

1. CREATING A NEW WIN32 APPLICATION¹

- ☐ Open Visual Studio 2010.
- ☐ In Visual Studio, click **File > New > Project** to open the New Project dialog.
- ☐ In the list on the left, under Installed Templates, select **Visual C++ > Win32**.
- ☐ In the center panel, select **Win32 Project**.
- ☐ In the **Name** text box, enter `Win32Asm`
- ☐ In the **Location** text box, select a location on a *non-network* drive (such as your local documents folder – `C:\Users\yourid\Documents\` – or a folder on a USB drive). *Your desktop is on the network—don't use that.*
- ☐ Click **OK**. The Win32 Application Wizard dialog will open.
- ☐ Click **Next** to proceed to the next page of the wizard.
- ☐ Under **Additional options**, check **Empty Project**.
- ☐ Click **Finish** to close the wizard.
- ☐ In the Solution Explorer (on the right side of the Visual Studio window), right-click on the Win32ConsoleAsm project, and from the context menu, select **Build Customizations**. This will open the Visual C++ Build Customization Files dialog.
- ☐ Check the box labeled **masm(.targets, .props)**.
- ☐ Click **OK** to close the dialog.

¹ The instructions here are partially based on <http://scriptbucket.wordpress.com/2011/10/19/setting-up-visual-studio-10-for-masm32-programming/>
A StackOverflow discussion of compiling assembly language code in Visual Studio 2010 is at <http://stackoverflow.com/questions/4548763/compiling-assembly-in-visual-studio>
which in turn references instructions for configuring build customizations at <http://connect.microsoft.com/VisualStudio/feedback/details/538379/adding-macro-assembler-files-to-a-c-project>

- ☐ In the Solution Explorer, right-click on **Source Files**, and select **Add > New Item**. This will open the Add New Item dialog.
- ☐ In the list on the left, select **Visual C++**.
- ☐ In the center panel, choose **Text File (.txt)**.
- ☐ In the **Name** text box, enter `main.asm`
- ☐ Click **Add** to close the dialog.
- ☐ Type the following into the `main.asm` file, and then save it.

```
.586
.model flat, stdcall
option casemap:none

ExitProcess PROTO, dwExitCode:DWORD

.code
main PROC
    push 0
    call ExitProcess
main ENDP
END main
```

- ☐ In the Solution Explorer, right-click the `Win32ConsoleAsm` project, and select **Properties** from the context menu.
- ☐ In the tree on the left, select **Configuration Properties > Linker > General**.
- ☐ In the list on the right, change the value of **Enable Incremental Linking** to *No (/INCREMENTAL:NO)*.
- ☐ In the tree on the left, select **Configuration Properties > Linker > Advanced**.
- ☐ In the list on the right, click on the text area to the right of **Entry Point**. Type `main` and press the Enter key. (This tells the linker that the entrypoint for the generated executable file will be a procedure named `main`. This must match the name in the `END` directive at the end of your `.asm` file.)
- ☐ Click **OK** to close the dialog.
- ☐ Click **Build > Build Solution**. (If the linker crashes, click **Build > Clean Solution**, then click **Build > Build Solution** again; usually, it will succeed the second time even if it crashes the first time.)
- ☐ Start debugging. Your program does not do anything yet, so your program should start and then terminate immediately.

2. DISPLAYING A MESSAGE BOX

- ☐ Change the contents of `main.asm` to the following, and save it. (Changes from before are in bold.)

```

.586
.model flat, stdcall
option casemap:none

ExitProcess PROTO, dwExitCode:DWORD


MessageBoxA PROTO,
    hWnd:DWORD,
    lpText:PTR BYTE,
    lpCaption:PTR BYTE,
    uType:DWORD

.data
mbMessage BYTE "This is a message box!", 0
mbTitle   BYTE "Title Goes Here", 0

.code
main PROC
    push 40h      ; What does this magic number do?
    push OFFSET mbTitle
    push OFFSET mbMessage
    push 0
    call MessageBoxA


    push 0
    call ExitProcess
main ENDP
end main

```

- ☐  Build and run your project. A message box should be displayed. What icon (image) is displayed in the message box? _____

Note that your program is invoking a function called `MessageBoxA`. There is also a `MessageBoxU` function. The former uses the ASCII/ANSI character set for strings; the latter uses Unicode. Microsoft's documentation simply refers to the two collectively as "the *MessageBox* function." A and U suffixes are used throughout the Win32 API to distinguish ANSI vs. Unicode versions of a function.

When you called the *MessageBox* function (or rather, *MessageBoxA*), you passed 40h for the *uType* parameter. This caused it to display the icon that you identified above. However, you typically would not pass a magic number like 40h directly. Instead, you would use a symbolic constant named `MB_ICONsomething`.

- ☐ Open Internet Explorer, and find the documentation for the *MessageBox* function on MSDN. (Perhaps, Google "messagebox function windows".)
- ☐  Skim the MSDN documentation for the *MessageBox* function. There are two different `MB_ICONxxx` constants that both correspond to the icon in your program (40h). What are they?
- `MB_ICON_____` and `MB_ICON_____`

- ☐ Change your assembly code to use one of these symbolic constants instead of the magic number 40h. You will need to add an `=` or `EQU` directive defining the `MB_ICONxxx` constant.

3. CREATE A WINDOW

- ☐ Change the contents of `main.asm` to the following, and save it. To avoid errors, you may want to download this code from Canvas instead of typing it; it is available as *lab6-part3.asm*.

```

.586
.model flat, stdcall
option casemap:none

; Win32 API constants
; =====

MB_ICONERROR = 10h
MB_ICONINFORMATION = 40h
CW_USEDEFAULT = 80000000h
SW_SHOWNORMAL = 1
WS_OVERLAPPEDWINDOW = 0CF0000h

; Win32 API functions
; =====

ExitProcess PROTO, dwExitCode:DWORD

GetModuleHandleA PROTO,
    lpModuleName:DWORD

RegisterClassExA PROTO,
    lpwcx:DWORD

DefWindowProcA PROTO,
    hWnd:DWORD,
    msg:DWORD,
    wParam:DWORD,
    lParam:DWORD

CreateWindowExA PROTO,
    dwExStyle:DWORD,
    lpClassName:DWORD,
    lpWindowName:DWORD,
    dwStyle:DWORD,
    x:DWORD,
    y:DWORD,
    nWidth:DWORD,
    nHeight:DWORD,
    hWndParent:DWORD,
    hMenu:DWORD,
    hInstance:DWORD,
    lpParam:DWORD

ShowWindow PROTO,
    hWnd:DWORD,
    nCmdShow:DWORD

MessageBoxA PROTO,
    dWnd:DWORD,
    lpText:PTR BYTE,
    lpCaption:PTR BYTE,
    uType:DWORD

.data
szClassName BYTE "sampleWindow", 0
szTitle     BYTE "COMP 3350 App", 0
szError     BYTE "Error", 0

```

```

WNDCLASS_START = $
cbSize DWORD WNDCLASS_END - WNDCLASS_START
style     DWORD 0
lpfnWndProc  DWORD OFFSET DefWindowProcA
cbClsExtra  DWORD 0
cbWndExtra  DWORD 0
hInstance   DWORD 0
hIcon       DWORD 0
hCursor     DWORD 0
hbrBackground  DWORD 6
lpzMenuName  DWORD 0
lpzClassName  DWORD OFFSET szClassName
hIconSm     DWORD 0
WNDCLASS_END = $

.code
main PROC
    ; Get the instance handle
    push 0
    call GetModuleHandleA
    mov hInstance, eax

    ; Register a window class
    push WNDCLASS_START
    call RegisterClassExA
    cmp eax, 0
    je error

    ; Create a window
    push 0
    push hInstance
    push 0
    push 0
    push 100
    push 500
    push CW_USEDEFAULT
    push CW_USEDEFAULT
    push WS_OVERLAPPEDWINDOW
    push OFFSET szTitle
    push OFFSET szClassName
    push 0
    call CreateWindowExA
    cmp eax, 0
    je error

    ; Show the window
    push SW_SHOWNORMAL
    push eax
    call ShowWindow

    ; FIXME: ADD MESSAGE LOOP HERE

    ; Exit cleanly
    push 0
    call ExitProcess

error:
    ; Display a generic error message
    push MB_ICONERROR
    push OFFSET szError
    push OFFSET szError
    push 0
    call MessageBoxA
    push 1
    call ExitProcess
main ENDP
END

```

- ☐ Build and run your project. It should start and then exit immediately. Even though you are invoking methods to create and display a window, you will probably not see the window yet.

The code is a bit long, but it is not complicated.

- **At the beginning of the file are Win32 constants and function prototypes.** When you code against the Win32 API in C or C++, you can simply `#include <windows.h>`, and these will be available to your program. In our assembly language program, we are defining them explicitly.
- **The .data section begins with normal data declarations,** specifically, three null-terminated strings.
- **The next twelve declarations define a *structure*.** In C/C++, the Win32 API defines a `WNDCLASSEX` structure as follows. The twelve declarations in the assembly code mimic the layout of this C/C++ structure.

```
typedef struct {
    UINT        cbSize;
    UINT        style;
    WNDPROC      lpfnWndProc;
    int         cbClsExtra;
    int         cbWndExtra;
    HINSTANCE    hInstance;
    HICON        hIcon;
    HCURSOR      hCursor;
    HBRUSH       hbrBackground;
    LPCTSTR      lpstrMenuName;
    LPCTSTR      lpstrClassName;
    HICON        hIconSm;
} WNDCLASSEX
```

- The *cbSize* member defines the total number of bytes in the structure. In the assembly code, we compute this using the current location counter (\$).
- The *lpstrClassName* member contains the memory address of a null-terminated string. In the assembly code, we compute this using the `OFFSET` operator.
- **The main procedure is essentially a sequence of five API calls:**
 1. **GetModuleHandleA.** This returns a handle for the process, which is used in several subsequent API calls.
 2. **RegisterClassExA.** Before your program can display a window, it must “register” a *window class*. This determines what icon the window will use, what its background color will be, what function will be used to process mouse and keyboard events for the window, etc. (Each of these is specified by one of the members of the `WNDCLASSEX` structure above.)
 3. **CreateWindowExA.** After the window class has been registered, this API call creates a new window. It returns a handle to the newly-created window (`hWnd`).
 4. **ShowWindow.** The newly-created window is initially invisible; this call makes the window appear.
 5. **ExitProcess** terminates the program.
 6. Both `RegisterClassExA` and `CreateWindowExA` return 0 if an error is encountered. If this occurs, we display a (very unhelpful) message box and exit with a nonzero exit code.

3. ADDING A MESSAGE LOOP

Currently, your program invokes the correct API functions to create and display a window, but the program does not appear to do anything. The problem is that you need to add a *message loop*.

- ☐ **Google “about messages and message queues” to find the MSDN page titled “About Messages and Message Queues (Windows)”.** The page’s text should start with, “Unlike MS-DOS-based applications, Windows-based applications are event-driven....” Read the first couple of paragraphs at the top of the page, then scroll down and read the section titled “Windows Messages.”

There are many messages that Windows sends to your application. For example:

- The WM_LBUTTONDOWN message indicates that the left mouse button was pushed down.
- The WM_CLOSE message indicates that a window’s Close button (X) was clicked, or Close was selected from the system menu in its upper-left corner.
- The WM_PAINT message indicates that a window needs to be redrawn (e.g., if another window was obscuring it but was moved out of the way, so the window is now visible).

Typically, each window class has its own function to handle messages for that specific (type of) window. The *message loop* is a *while* loop in the program’s *main* procedure that receives messages from Windows and passes them on to the event handlers for individual windows.

In the remaining steps, you will add a message loop to your program.

- ☐ **Add prototypes for GetMessageA, TranslateMessage, and DispatchMessageA.**

```
GetMessageA PROTO,  
    lpMsg:DWORD,  
    hWnd:DWORD,  
    wParamFilterMin:DWORD,  
    wParamFilterMax:DWORD  
  
TranslateMessage PROTO,  
    lpMsg:DWORD  
  
DispatchMessageA PROTO,  
    lpMsg:DWORD
```

- ☐ **Reserve space in your .data section to store a message.** You will not need to access the individual parts of a message, so you can simply allocate 28 bytes. Add this to your .data section:

```
msg BYTE 28 DUP(?)
```

In C/C++, you would declare a variable of type MSG. Like WNDCLASSEX, this is a structure type. Our assembly language program will not need to access the individual parts of a message, so we can just allocate 28 bytes instead of declaring each individual member of the message structure.

- ☐ **Add the message loop after the call to ShowWindow, before the call to ExitProcess.** In C/C++, the message loop would look something like this:

```
while (GetMessageA(&msg, 0, 0, 0) != 0) {  
    TranslateMessage(&msg);  
    DispatchMessageA(&msg);  
}
```

Observe what this loop does: it invokes *GetMessageA*, *TranslateMessage*, and *DispatchMessageA* repeatedly, stopping as soon as *GetMessageA* returns 0.

Translate this *while* loop into assembly language.

Note that the *address* of `msg` is being passed to each function. So, for example, the call to *GetMessageA* should be translated as

```
push 0  
push 0  
push 0  
push OFFSET msg  
call GetMessageA
```

- ☐ **Build and run your project.** It should display a window; you should be able to resize and close the window.
- ☐ When you close the window, your program should continue running, even though nothing is visible. Switch back to Visual Studio and terminate your program.

Save your code! You will need it on Wednesday. We will fix this problem—we'll make your program terminate when the window is closed—and we'll make the window slightly more interesting...