Tutorial 2: MessageBox

In this tutorial, we will create a fully functional Windows program that displays a message box saying "Win32 assembly is great!".

Download the example file here.

Theory:

Windows prepares a wealth of resources for Windows programs. Central to this is the Windows API (Application Programming Interface). Windows API is a huge collection of very useful functions that reside in Windows itself, ready for use by any Windows programs. These functions are stored in several dynamic-linked libraries (DLLs) such as kernel32.dll, user32.dll and gdi32.dll. Kernel32.dll contains API functions that deal with memory and process management. User32.dll controls the user interface aspects of your program. Gdi32.dll is responsible for graphics operations. Other than these "main three", there are other DLLs that your program can use, provided you have enough information about the desired API functions.

Windows programs dynamically link to these DLLs, ie. the codes of API functions are not included in the your program's executable file. In order for your program to know where to find the desired API functions at runtime, you have to embed that information into the executable file. The information is in import libraries. You must link your programs with the correct import libraries or they will not be able to locate API functions.

When a Windows program is loaded into memory, Windows reads the information stored in the program. That information includes the names of functions the program uses and the DLLs those functions reside in. When Windows finds such info in the program, it'll load the DLLs and perform function address fixups in the program so the calls will transfer control to the right function.

There are two categoriesof API functions: One for ANSI and the other for Unicode. The names of API functions for ANSI are postfixed with "A", eg. MessageBoxA. Those for Unicode are postfixed with "W" (for Wide Char). Windows 95 natively supports ANSI and Windows NT Unicode.

We are usually familiar with ANSI strings, which are arrays of characters terminated by NULL. ANSI character is 1 byte in size. While ANSI code is sufficient for European languages, it cannot handle several oriental languages which have several thousands of unique characters. That's why UNICODE comes in. A UNICODE character is 2 bytes in size, making it possible to have 65536 unique characters in the strings.

HLA and the HLA Standard Library use ANSI characters by default. It is possible to write HLA programs that utilize UNICODE (HLA, after all, is assembly language and you can do *anything* with assembly language), but you will not be able to use many of the HLA Standard Library routines if you use UNICODE. Hence this tutorial will stick to ANSI characters.

Example:

I'll present the bare program skeleton below. We will flesh it out later.

```
program tut2;
begin tut2;
end tut2;
```

The execution starts from the first instruction immediately below the “begin tut2” clause. The execution will proceed instruction by instruction until some flow-control instructions such as jmp, jne, je, ret etc is found. Those instructions redirect the flow of execution to some other instructions. When the program encounters the “end tut2” clause, it automatically exits to Windows (this is because HLA automatically emits code at the end of the program to do this for you). Should you wish to prematurely exit the program, you have three options: (1) you can JMP to the bottom of the program; (2) you can execute the HLA high level “EXIT tut2;” statement; or (3) you can directly call the Windows API routine **ExitProcess**. This last option is especially useful if you wish to return a process “return code” to the operation system. By default, HLA programs always return a zero return code (that signifies “no error”) when they return via EXIT or by running of the end of the main program. If you wish to return an error status (non-zero) value when your program quits, you will need to call **ExitProcess** directly. You can directly call ExitProcess in your programs by defining the following prototype in your static **section**:

```
static
  ExitProcess: procedure( uExitCode:uns32 );
                external( “__imp__ExitProcess@4” );
```

The above line is called a procedure pointer prototype (“__imp__ExitProcess@4” is actually a pointer to the function, not the name of the function itself). A procedure pointer prototype defines the attributes of a function to the assembler/linker so it can do type-checking for you.

In short, the name of the procedure followed by a colon and the keyword **procedure** and then by the list of parameters and their data types separated by commas. In the **ExitProcess** example above, it defines **ExitProcess** as a procedure which takes only one parameter of type **uns32**. Procedure prototypes are very useful when you use the high-level call syntax. For example, if you do:

```
call ExitProcess
```

without pushing an uns32 value onto the stack, the assembler/linker will not be able to catch that error for you. You'll notice it later when your program crashes. But if you use:

```
ExitProcess();
```

the linker will inform you that you forgot to push a dword on the stack thus avoiding error. I recommend you use invoke instead of simple call. The high level calling syntax is as follows:

```
Procname( optional_arguments );
```

Procname can be the name of a procedure or it can be a procedure pointer. The procedure parameters are separated by commas.

Most of prototypes for API functions are kept in include files. This normally appear in the “c:\hla\include” subdirectory (or wherever you’ve installed HLA). The include files have “.hhf” (HLA Header File) extensions and the prototypes for the functions in a DLL are generally found in the “win32.hhf” header file. For example, **ExitProcess’** prototype is is in the win32.hhf file. There are a few differences between HLA’s win32 API definitions and the standard Windows definitions. For efficiency reasons, HLA tends to place function prototypes in *namespaces*. This saves memory and time during compilation. Most of the win32 API functions are members of the :”win” namespace, so you would actually call the function using the following syntax:

```
win.ExitProcess( return_code );
```

This document will point out other differences between Windows and HLA as necessary.

You can also create procedure prototypes for your own procedures.

Now back to **ExitProcess**, the **uExitCode** parameter is the value you want the program to return to Windows after the program terminates. You can call *ExitProcess* like this:

```
win.ExitProcess( 0 );
```

Put that line immediately below the “begin tut2;” clause and you will get a win32 program which immediately exits to Windows, but it's a valid program nonetheless (of course, HLA automatically emits a call to win.ExitProcess when it encounters the “end tut2;” clause, so there’s no need to manually insert this call yourself, but the call is legal).

```
program tut2;
#include( “win32.hhf” )
begin tut2;

  win.ExitProcess( 0 );

end tut2;
```

Note the new statement, **#include**. This statement has an operand that is the name of a file you want to insert at the place the statement appears. In the above example, when HLA processes the line #include(“win32.hhf”), it will open win32.hhf which is in the “hla\include” folder and processes the content of win32.hhf as if you had pasted the contents of that file there. HLA's win32.hhf file contains procedure prototypes and definitions of constants and structures you need in win32 programming. The include files save you the work of typing out the prototypes yourself so use them whenever you can.

Now save the example under the name msgbox.hla. Assuming that hla.exe is in your path, compile msgbox.hla with:

hla msgbox.hla

After you successfully compile msgbox.hla, you will get msgbox.exe.Now that you have msgbox.exe. Go on, run it. You'll find that it does nothing. Well, we haven't put anything interesting into it yet. But it's a Windows program nonetheless. And look at its size! In my PC, it is 20K bytes.

Next we're going to put in a message box. Its function prototype is:

```
static
  MessageBox:procedure
             (
                uType:uns32;
                lpCaption:string;
                lpText:string;
                hwnd:dword
              ) external( “__imp__MessageBoxA@16” );
```

As noted in tutorial #1, the parameters are reversed compared to the standard Windows definition because HLA uses the Pascal calling sequence rather than the STDCALL calling sequence. This is very easily corrected by defining a macro that reverses the parameters when actually calling MessageBox.

The **hwnd** parameter is the handle to parent window. You can think of a handle as a number that represents the window you're referrring to. Its value is not important to you. You only remember that it represents the window. When you want to do anything with the window, you must refer to it by its handle. For our simple example there will be no parent window, so we'll just supply NULL (0) as the value for this parameter.

lpText is a pointer to the text you want to display in the client area of the message box. A pointer is really an address of something. In HLA, string variables are also pointers. Hence we can simply give this parameter the type string and let HLA take care of the details concerning address computation of the string. MessageBox requires a pointer to a sequence of characters that end with a zero byte. HLA's strings are actually a bit more sophisticated that this, but the pointer held in a string variable does point at a zero-terminated sequence of characters, so we can use HLA strings as-is whenever a Windows API function expects a pointer to a zero terminated string.

lpCaption is a pointer to the caption of the message box. This is also a string parameter so we can use the HLA string type for this parameter.

uType specifies the icon and the number and type of buttons on the message box. This particular value is specified via some combination of the **win.MB_*xxxx*** constants that appear in the win32.hhf header file. For the time being, we'll use the value **win.MB_OK** which displays a single "Okay" button in the dialog box.

Let's modify msgbox.hla to include the message box.

```
 // Iczelion's tutorial #2: MessageBox

program msgBoxDemo;
#include( "win32.hhf" )

begin msgBoxDemo;

    win.MessageBox
    (
        0,                              // NULL hWnd if no owner.
        "Iczelion's tutorial no.2",     // Window title.
        "Win32/HLA Assembly is Great!", // Text to display in window.
        win.MB_OK                       // Display an "OK" button.
    );

end msgBoxDemo;
```

Assemble and run it. You will see a message box displaying the text "Win32 Assembly is Great!".

Let's look again at the source code.

We use the constant win.MB_OK. This constant is defined in win32.hhf. So you can refer to it by name instead of its numeric value. This improves readability of your source code.