Randy Hyde's Win32 Assembly Language Tutorials (Featuring HOWL)

#1: Hello World

Prerequisites:

This tutorial set assumes that the reader is already familiar with assembly language programming and HLA programming in particular. If you are unfamiliar with assembly language programming or the High Level Assembler (HLA), you will want to grab a copy of my book "The Art of Assembly Language, 2nd Edition" from No Starch Press (www.nostarch.com). The HOWL (HLA Object Windows Library) also makes heavy use of HLA's object-oriented programming facilities. If you are unfamiliar with object-oriented programming in assembly language, you will want to check out the appropriate chapters in "The Art of Assembly Language" and in the HLA Reference Manual. Finally, HOWL is documented in the HLA Standard Library Reference Manual; you'll definitely want to have a copy of the chapter on HOWL available when working through this tutorial.

Source Code:

The source code for the examples appearing in this tutorial are available as part of the HLA Examples download. You'll find the sample code in the Win32/HOWL subdirectories in the unpacked examples download. This particular tutorial uses the files *001_helloworld.hla* and *001x_helloworld.hla*.

Hello World:

There is a long-standing tradition (since the days of K&R C) of providing a "Hello World" program as the first programming example when presenting a new language or application framework. Printing the string "Hello World" is often a trivial example that requires a minimal amount of programming language knowledge to comprehend. The purpose of a "Hello World" example, of course, is not to teach someone how to print the string "Hello World" to a display (or wherever else the output might go) but, rather, to teach the reader all the steps needed to get a working program to the point you can execute it. Because this is important knowledge that someone will need in order to create Win32 applications using the HOWL application framework, "Hello World" seems like a reasonable place to start our journey.

Many Win32 programming tutorials actually begin by teaching you how to print the string "Hello World" in the middle of a form; however, I feel that such an example is needlessly overkill. Not that putting the string "Hello World" on a form is difficult when using HOWL, but the truth is that if we can get a window to appear on the display at all, we've achieved everything the "Hello World" program was intended to achieve. Don't worry, though, you'll get to see "Hello World" on the screen -- we'll sneak that string in as the window's title.

Perhaps the best place to start is by describing the HOWL (HLA Object Windows Library) philosophy. HOWL is an *application framework*. This means that HOWL is largely a pre-canned application to which you add code to differentiate it from other applications. If you're already familiar with Win32 API programming, you're probably aware of how much work it is to write a trivial Windows program (message handling loops, decoding messages, stuff like that). The HOWL library encapsulates all that functionality so you can concentrate on writing only the code you need to implement the functionality of your application.

HOWL is based on two HLA facilities: object-oriented programming and macros. The existing HOWL library code is written using HLA class facilities. Win32 GUI programming lends itself to an object implementation and this is exactly what the HOWL library provides -- an object-oriented view of the Win32 API. The second big part of the HOWL system is the HOWL Declarative Language. This is a very high-level programming language, implemented with HLA's macros, that lets you define the look and feel of a Win32 GUI app by simply declaring the objects that will appear on a form (window).

A typical (small) HOWL application will consist of two parts: one section where you define the form layout (using the HOWL Declarative Language) and a second part containing mostly boilerplate code that you usually copy from application to application.

For our "Hello World" application, here's what the first section looks like (we'll discuss these statements after the code:

```
program helloworld;
#linker( "comdlg32.lib" )
#linker( "comctl32.lib" )

?@NoDisplay     := true;
?@NoStackAlign  := true;

#includeOnce( "stdlib.hhf" )
#includeOnce( "howl.hhf" )


const
    applicationName := "Hello World";
    formX           := w.CW_USEDEFAULT; // Let Windows position this guy
    formY           := w.CW_USEDEFAULT;
    formW           := 600;
    formH           := 600;




// Form declaration.
// We've got an empty form with no widgets, so this is fairly trivial:

wForm( mainAppWindow );
endwForm


// Must include the following macro invocation to emit the code that the
// wForm macro generated:
```

```
mainAppWindow_implementation();



// The following gets called immediately after the main application
// window is created. It must be provided, even if it does nothing.

method mainAppWindow_t.onCreate;
begin onCreate;
end onCreate;
```

The first two statements of interest are the `#linker` statements:

```
#linker( "comdlg32.lib" )
#linker( "comctl32.lib" )
```

The HOWL library uses code in the Win32 common dialog (comdlg32.lib) and common control (comctl32.lib) libraries. Normally, an HLA compilation doesn't automatically link these libraries into your program code. You could manally specify these library names on the command-line when you compile a HOWL application, but it's much easier to simply include these two #linker commands at the beginning of your source file.

```
?@NoDisplay     := true;
?@NoStackAlign  := true;
```

By default, HLA automatically emits some extra code at the beginning of most procedures to build a display and to ensure that the stack is aligned to four bytes. This is done for beginning assembly language students; more advanced assembly language programmers (and this would include anyone working through these tutorials) won't want this "coding behind your back" to take place. Therefore, the two statements above appear in most HLA programs (unless you really intend to use procedure displays in your code, in which case I presume you know why you wouldn't want to include the statement above). Windows requires all stacks to be dword aligned before making any Win32 API calls, so it would seem to be a good thing to have HLA automatically dword align the stack. However, you have to go out of your way to misalign the stack; I'm assuming, because you're reading this, that you realize that you should always keep the stack dword-aligned in your code (by always pushing or popping objects that are a multiple of four bytes long).

```
#includeOnce( "stdlib.hhf" )
#includeOnce( "howl.hhf" )
```

Almost every HLA application is going to include the HLA standard library header files. HOWL applications also require that you include *howl.hhf*. Note that the *howl.hhf* header file also includes the *w.hhf* (Windows) header file, so you don't have to explicitly include *w.hhf* to gain access to Windows constants and data structures. However, this may change in the future, so

if you want to "future-proof" your programs, you may want to consider adding a `#include-Once("w.hhf")` statement to the above sequence.

```
const
    applicationName := "Hello World";
    formX           := w.CW_USEDEFAULT; // Let Windows position this guy
    formY           := w.CW_USEDEFAULT;
    formW           := 600;
    formH           := 600;
```

As noted earlier, the second part of a typical HOWL application is boiler-plate code that generally remains unchanged from application to application. These five constant declarations are essentially *parameters* to that boilerplate code. The applicationName constant is the window title that Windows will dislay in the title bar of the main form (window) that the application creates. This particular string is where we'll put "Hello World", to justify calling the program a "Hello World" program.

The `formX, formY, formW`, and `formH` constants define the position and size of the form on the screen. The Windows' constant `w.CW_USEDEFAULT` instructs Windows to use a value that it feels is most appropriate. In the case of the (x,y) coordinate values, Windows will pick a spot on the screen that varies each time you run an application so that the windows are staggered on the display. Almost all of the applications in this series will use a 600x600 pixel size. Most will work will with an even smaller form. You can play around with these numbers if you like; just keep in mind that many examples (Hello World excluded) are going to place a single button in the lower-right hand corner of the window, so take this into account if you decide to change the window size. Of course, for the Hello World application, we're not going to put anything into the window, so as long as the window is wide enough to display "Hello World" in the title bar, you should be fine.

```
// Form declaration.
// We've got an empty form with no widgets, so this is fairly trivial:

wForm( mainAppWindow );
endwForm


// Must include the following macro invocation to emit the code that the
// wForm macro generated:

mainAppWindow_implementation();
```

Now we get to the HOWL part of the program. Specifically, `wForm..endwForm` are statements in the HOWL Declarative Langage (HDL) that let you define what your application's main window is going to look like. The `wForm` statement contains a single argument -- a form name. This must be a valid (and unique) HLA identifier. The `wForm` statement creates several identifiers from this argument you pass to `wForm`:

- `mainAppWindow_t`: an HLA class type derived from the HOWL `wForm_t` class type. The `wForm` macro automatically generates the class definition for this type based on the source code appearing between the `wForm` and `endwForm` statements.
- `mainAppWindow_p`: an HLA type that is a pointer to an object of type `mainAppWindow_t`.
- `mainAppWindow`: a variable (object) of type `mainAppWindow_t`.
- `pmainAppWindow`: a variable of type `mainAppWindow_p` that will contain a pointer to the object that the `wForm..endwForm` macro invocation creates. Specifically, the HOWL Declarative Language will initialize this variable with the address of the `mainAppWindow` variable.
- `mainAppWindow_implementation`: a macro that you must explicitly invoke to create the constructor for the class.

The `wForm..endwForm` sequence actually creates two things: the variables and types just mentioned and a macro named `mainAppWindow_implementation`. The `wForm..endwForm` sequence is actually equivalent to a class declaration in HLA (indeed, that's what this macro directly expands into). However, while generating the `mainAppWindow_t` class (or whatever class name you specify as the `wForm` argument), the `wForm` macro is also busy writing a macro that will expand into the constructor for the new class. For technical reasons (specifically: the `wForm..endwForm` sequence might be in a header file), HOWL does not automatically expand this macro. The last statement in the above code sequence does this expansion and generates the constructor for the `mainAppWindow_t` class.

```
// The following gets called immediately after the main application
// window is created. It must be provided, even if it does nothing.

method mainAppWindow_t.onCreate;
begin onCreate;
end onCreate;
```

The last bit of code in the first part of our Hello World application is the definition of the `mainAppWindow_t` class' `onCreate` method. HOWL will call this method immediately after creating the main form's window. Normally, you'd put initialization code in here that must execute after the form is created but before the application framework takes over. For our simple Hello World application, no such initialization is necessary, but because the `onCreate` method has been defined for the class, we must still provide the method (even if it is empty and does nothing but return to the caller).

The second part of a HOWL application, the boilerplate code, is generally the following:

```
////////////////////////////////////////////////////////////////////////
//
//
// The following is mostly boilerplate code for all apps (about the only thing
// you would change is the size of the main app's form)
//
//
////////////////////////////////////////////////////////////////////////
//
// When the main application window closes, we need to terminate the
// application. This overridden method handles that situation.
```

```
method mainAppWindow_t.onClose;
begin onClose;

    // Tell the winmain main program that it's time to terminate.
    // Note that this message will (ultimately) cause the appTerminate
    // procedure to be called.

    w.PostQuitMessage( 0 );


end onClose;


// When the application begins execution, the following procedure
// is called.  This procedure must create the main
// application window in order to kick off the execution of the
// GUI application:

procedure appStart;
begin appStart;

    push( esi );

    // Create the main application window:

    mainAppWindow.create_mainAppWindow
    (
        applicationName,        // Window title
        w.WS_EX_CONTROLPARENT,  // Need this to support TAB control selection
        w.WS_OVERLAPPEDWINDOW,  // Style
        NULL,                  // No parent window
        formX,                  // x-coordinate for window.
        formY,                  // y-coordinate for window.
        formW,                  // Width
        formH,                  // Height
        howl.bkgColor_g,        // Background color
        true                    // Make visible on creation
    );
    mov( esi, pmainAppWindow ); // Save pointer to main window object.
    pop( esi );

end appStart;



// appTerminate-
//
//  Called when the application is quitting, giving the app a chance
// to clean up after itself.
//
// Note that this is called *after* the mainAppWindow_t.onClose method
// executes (indeed, mainAppWindow_t.onClose, by posting the quit message,
// is what actually causes the program to begin terminating, which leads
// to the execution of this procedure).

procedure appTerminate;
begin appTerminate;
```

```
    // Clean up the main application's form.
    // Note that this will recursively clean up all the widgets on the form.

    mainAppWindow.destroy();

end appTerminate;


// appException-
//
// Gives the application the opportunity to clean up before
// aborting when an unhandled exception comes along:

procedure appException( theException:dword in eax );
begin appException;

    raise( eax );

end appException;



// The main program for a HOWL application must simply
// call the HowlMainApp procedure.

begin helloworld;

    // Set up the background and transparent colors that the
    // form will use when registering the window_t class:

    w.GetSysColor( w.COLOR_MENU );
    mov( eax, howl.bkgColor_g );
    or( $FF00_0000, eax );
    mov( eax, howl.transparent_g );
    w.CreateSolidBrush( howl.bkgColor_g );
    mov( eax, howl.bkgBrush_g );

    HowlMainApp();

    // Delete the brush we created earlier:

    w.DeleteObject( howl.bkgBrush_g );

end helloworld;
```

Again, we'll discuss all this code on a piece-by-piece basis:

```
method mainAppWindow_t.onClose;
begin onClose;

    // Tell the winmain main program that it's time to terminate.
    // Note that this message will (ultimately) cause the appTerminate
    // procedure to be called.

    w.PostQuitMessage( 0 );
```

```
end onClose;
```

The `onClose` method terminates the application. The call to `w.PostQuitMessage` is a magic Windows API call that tells the application to terminate itself. You can tell your application to terminate at anytime by directly invoking this method. You can do this with the following statement:

```
mainAppWindow.onClose();
```

Note that this function will return to the caller. Specifically, your program does not immediately quit when you execute this statement. Instead, it will quit at some point in the near future when Windows gets around to processing the quit message. Therefore, your code should be ready to continue execution upon return from the method call above.

```
procedure appStart;
begin appStart;

    push( esi );

    // Create the main application window:

    mainAppWindow.create_mainAppWindow
    (
        applicationName,        // Window title
        w.WS_EX_CONTROLPARENT,  // Need this to support TAB control selection
        w.WS_OVERLAPPEDWINDOW,  // Style
        NULL,                  // No parent window
        formX,                  // x-coordinate for window.
        formY,                  // y-coordinate for window.
        formW,                  // Width
        formH,                  // Height
        howl.bkgColor_g,        // Background color
        true                    // Make visible on creation
    );
    mov( esi, pmainAppWindow ); // Save pointer to main window object.
    pop( esi );

end appStart;
```

The HOWL framework calls `appStart` after it has completed various Windows' initialization operations. The main purpose of the appStart procedure is to call the constructor for the application's main form (window). Assuming you stick with the name "mainAppWindow" in your `wForm` statement, the code above is very typical for the `appStart` procedure. Let's consider the statements in this procedure on a section-by-section basis:

```
    mainAppWindow.create_mainAppWindow
    (
        applicationName,        // Window title
        w.WS_EX_CONTROLPARENT,  // Need this to support TAB control selection
        w.WS_OVERLAPPEDWINDOW,  // Style
        NULL,                  // No parent window
        formX,                  // x-coordinate for window.
        formY,                  // y-coordinate for window.
```

```
        formW,                  // Width
        formH,                  // Height
        howl.bkgColor_g,        // Background color
        true                    // Make visible on creation
    );
    mov( esi, pmainAppWindow ); // Save pointer to main window object.
```

The remaining (interesting) code in the `appStart` procedure is a call to the `mainAppWindow` constructor that initializes the `mainAppWindow` variable that the `wForm..endwForm` sequence creates. Before describing the arguments to the constructor's call, it's important to point out that if you use a name other than "mainAppWindow" in the `wForm` statement, then you will need to adjust the object name in this procedure invocation accordingly. This is why the appStart code isn't hidden away in a library module somewhere. If you want to use a name other than `mainAppWindow` for your form's name (or if you want to use more than one form in an application), you will need to modify `appStart` accordingly.

The first argument tothe constructor, `applicationName`, is the string that HOWL will display in the title bar for the window. Using the applicationName string constant here lets you specify the actual string constant at the beginning of the source file ("Hello World" in this example).

The next two arguments are Windows' CreateWindowEx extended style and style values. the `w.WS_EX_CONTROLPARENT` extended-style argument tells HOWL that the user should be allowed to select different widgets on the form using the tab key. If you don't want to allow this, just put the value zero here. The `w.WS_OVERLAPPEDWINDOW` CreateWindow style is the typical Windows style you'll want for a HOWL form. For more details on the available style and extended-style constants you can use here, please consult the documentation for the Windows' CreateWindow and CreateWindowEx API calls (a search on-line will provide this documentation if you don't have it handy).

The fourth parameter is the handle of the form's parent window. The main form has no parent window, so this value is always NULL. If you create multiple forms, some of those forms could reference the handle of the main form (e.g., `mainAppWindow.handle`) in this argument.

The next four arguments specify the position on the screen and the size of the form. These are the constants declared earlier in the source file.

The ninth argument is the RGB value of the background color to use when painting the client area of the form (in our case, the client area is everything below that title bar, not including the window's border). This example uses the `howl.bkgColor_g` value that is created in the main program (a little later).

The last argument is a boolean value that determines whether HOWL will create the form in a visible (true) or hidden (false) state. For the main form, this is always true. If you are creating multiple forms, you'll want to specify false here if you want to manually display those forms at some later point in your program.

The next piece of code in the HOWL application is the `appTerminate` procedure. HOWL calls `appTerminate` whenever the application is shutting down. This call is made after all the windows are closed and HOWL is done using the data objects you've created with a `wForm..endwForm` sequence. It is appTerminate's responsibility to clean up any system resources in use by the application.

```
procedure appTerminate;
begin appTerminate;

    // Clean up the main application's form.
    // Note that this will recursively clean up all the widgets on the form.

    mainAppWindow.destroy();

end appTerminate;
```

The `appTerminate` program in the boilerplate code does two things: first, it calls the destructor for the `mainAppWindow` object. This cleans up all objects placed on the form (deallocating memory and other system resources they use). It's always a good idea to free up system resources you use; anything you create in `appStart` should be destroyed or freed in the `appTerminate` procedure.

The HOWL system will call the `appException` procedure if an exception occurs. The default `appException` procedure in the boilerplate code simply re-raises this exception. You might want to modify this code to do any "last chance" exception handling (perhaps freeing up `howl.bkgBrush_g` might not be a bad idea, for example). As a general rule, you should attempt to catch exceptions closer to their source rather than placing a lot of exception handling code in the `appException` procedure. However, `appException` can be usefor as a catch-all for exceptions that would otherwise be missed.

The last bit of code in the HOWL boilerplate section is the main program. The main program is very simple, it just calls the `HowlMainApp` procedure:

```
begin helloworld;

    // Set up the background and transparent colors that the
    // form will use when registering the window_t class:

    w.GetSysColor( w.COLOR_MENU );
    mov( eax, howl.bkgColor_g );
    or( $FF00_0000, eax );
    mov( eax, howl.transparent_g );
    w.CreateSolidBrush( howl.bkgColor_g );
    mov( eax, howl.bkgBrush_g );

    HowlMainApp();

    // Delete the brush we created earlier:

    w.DeleteObject( howl.bkgBrush_g );

end helloworld;
```

The first set of statements set up the background color for use by the form and various controls. The `howl.bkgColor_g` and `howl.transparent_g` are RGB color values available for use by various components in the application. We'll take a look at the purpose of the `howl.transparent_g` value when we discuss transparency in the tutorial on graphic objects. The `howl.bkgBrush_g` is also available to the application if it wants to use it. Note that this code deletes the

brush before the application quits (it's always a good idea to free resources that your program uses). Technically, initializing these HOWL objects isn't strictly necessary; the HOWL library doesn't actually use them. However, their values are quite useful when creating HOWL applications, so the boilerplate HOWL main program always initializes them with a gray color. If you'd prefer a different background color, change the `w.GetSysColor` to load EAX with some appropriate RGB value. Though we'll discuss this farther in the tutorial on graphic objects, a background color whose H.O. byte contains $FF is a special "transparent" background color and for certain objects HOWL will not fill the background area of an object if the color is transparent. Normal RGB color values have a $00 value in the H.O. byte.

`HowlMainApp` is the real main program. But it appears in the HLA standard library, so it had to be made into a callable procedure to avoid linking errors. Therefore, in order to activate the HOWL application framework, the application's main program must call `HowlMainApp`.

Okay, we've described the code on a line-by-line basis. Now how do we compile and run it? Actually, there's not much special about compiling and running this app. From a command-line window, just type the following command (this assumes you're using the source file present in the HLA examples download from the win32\howl directory, which contains all the source code for this set of tutorials in an electronic form):

```
hla 001_HelloWorld
```

You can run the program by typing "001_HelloWorld" on the command line or by double-clicking on the executable's icon from the Windows Explorer.

If you double-click on the icon, you'll notice that the program opens up a console window in addition to the main form. Although the Hello World example doesn't use this window, most of the following tutorials will print debug and status information to this window. So generally, it's a good idea to leave this window up during the execution of these tutorial programs. However, for final releases of a GUI app, it's probably a good idea to banish this console window. You can do this by using the following command line to compile the program:

```
hla -w 001_HelloWorld
```

The HOWL Classes

Throughout this set of tutorials, I plan to discuss several of the HOWL classes (at least, as they exist at the time this tutorial was written; they are in constant flux so the details may have changed by the time you read this, but the main ideas should still apply). This will help familiarize you with the various HOWL classes at a reasonable pace.

The Hello World application indirectly[1] uses only one class: `wForm_t`. Here's the HLA definition for that class:

---

1. "Indirectly" is used here because the Hello World application actually creates a new class that is derived from `wForm_t` and uses that new class.

```
wForm_t:
    class inherits( window_t );
        var
            align( 4 );
            wForm_private:
                record

                    menuList    :wMenuItem_p;

                endrecord;

        procedure create_wForm
        (
            wwName      :string;
            caption     :string;
            exStyle     :dword;
            style       :dword;
            parent      :dword;
            x           :dword;
            y           :dword;
            width       :dword;
            height      :dword;
            fillColor   :dword;
            visible     :boolean
        );  external;

        method appendMenuItem( mi:wMenuItem_p );    external;


        override method insertWidget;               external;
        override method processMessage;             external;

    endclass;
```

There really isn't much to this class. All of the magic occurs in the `inherits( window_t );` clause (this means that the class actually includes many other data fields, procedures, methods, and iterators that `wForm_t` inherits from `window_t` and all the ancestor classes it inherits from).

The first thing of importance to note here is the `wForm_private` record variable. HLA does not provide "private" or "protected" data fields like some other object-oriented languages like C++ or Java. Because HLA is assemly language, any attempt to make a field private could be easily circumvented, so HLA doesn't even bother to try to hide private data fields. However, the fact that HLA doesn't bother with private data fields doesn't mean that the concept (information hiding) isnt' a good one. In HOWL, private data fields are "protected" by placing them in a record structure that has the name *`className`*`_private` (for example, wForm_private). Application programs should never access data in one of HOWL's private records. For those private data fields whose data is interesting to an application program, HOWL generally provides *accessor* functions (that return the value of a private data field) and *mutator* functions (that store a value into a private data field). Although using accessor and mutator functions is decidedly less convenient than reading and writing the data field directly, by channeling all private data access through these functions it is possible to automatically handle other chores (such a redrawing an object when you change its size or computing a Windows brush value when you change a color) that you would have to (remember to) take care of yourself. HOWL's mutator and accessor functions also help

maintain consistency between Windows' internal data structures and HOWL's data structures; something that wouldn't happen if you were to directly access the private data fields.

If some HOWL class' data field does not appear within a *className*_private record, then this is generally a public data field and (unless otherwise documented) you are free to read its value and, in most cases, write its value as well. A few fields are "read-only public." That means you can directly read their values but you should never write to them. Such fields do not appear within a *className*_private record structure, so you have to check the documentation on a public data field to ensure that it's okay to directly write to that data object. Fortunately, it's generally obvious from the use of a public data field whether you should be writing a value to it.

The wForm_t class has a single (private) data field: menuList. HOWL uses this field to create a linked list of menu items if your main form has a menu on it (Hello World does not). In general, you should ignore this field; if your form has menus, HOWL will automatically use this field; if your form does not have menus, HOWL doesn't use this field (and neither should you).

The appendMenuItem method attaches a single menu item (wMenuItem_t) to a form's menuList linked list. If you're using the HOWL Declarative Language (HDL) to create your forms, then you'll probably never directly call this method (the code that the HDL generates will call it for you). If you're masochistic enough to want to manually create a form with a menu attached to it (an example will appear in a later tutorial), then you can all this method after initializing the menu item to attach that menu item to the form.

A wForm_t object (or descendant object) is an example of a HOWL *container*. A container has the ability to hold (contain) other HOWL objects. For example, a form may contain buttons, edit boxes, labels, and other HOWL objects. The insertWidget method lets you place an object such as a button onto a form. If you dynamically create objects at run time, you'll attach them to a form by calling the insertWidget method.

The processMessage method is a private method that the HOWL system uses to process Windows' messages. Application programs must never call this method directly.

Another HOWL class worth discussing here is the wBase_t class. This is the base class for the HOWL system and, as such, all HOWL classes (including wForm_t) inherit all the fields and methods associated with wBase_t. Here is the wBase_t definition:

```
wBase_t:
    class

        var
            handle      :dword;
            _name       :string;
            wType       :lword;

            wBase_private:
                record

                    visible         :boolean;
                    enabled         :boolean;
                    onHeap          :boolean;
                    align( 4 );

                    objectID        :dword;
                    nextWidget      :wBase_p;
```

```
            // Pointer the wForm object that this
            // object belongs to.

            parentForm      :wForm_p;

            // Handle of the Windows parent window associated
            // with this control.  Note that parentForm.handle
            // may not be the same as parentHandle because this
            // object could belong to some other window that
            // is a child window of the main form. (Okay, parentForm
            // was probably a bad name to use).

            parentHandle    :dword;


        endrecord;


static
    objectID_g      :dword;     external( "objectID_object_t" );

// Constructors/Destructors:

procedure create_wBase
(
    wbName  :string
);  external;

method destroy;                                 external;
method show;                                    external;
method hide;                                    external;
method enable;                                  external;
method disable;                                 external;

// Accessor/mutator functions:

method get_handle;      @returns( "eax" );      external;
method get_objectID;    @returns( "eax" );      external;
method get_visible;     @returns( "al" );       external;
method get_enabled;     @returns( "al" );       external;
method get_onHeap;      @returns( "al" );       external;
method get_parentHandle;    @returns( "eax" );  external;
method get_parentForm;      @returns( "eax" );  external;

method set_onHeap( onHeap:boolean );            external;
method set_parentHandle( parentHandle:dword );  external;
method set_parentForm( parentForm:wForm_p );    external;


// Default message processor:

method processMessage
(
    hwnd    :dword;
    uMsg    :dword;
    wParam  :dword;
    lParam  :dword
);  external;
```

```
endclass;
```

This class has three public fields: `handle, _name`, and `wType`. The first public field, `handle`, is a read-only public field (you should never store a value into this field). Technically, `handle` should be a private field (there is even an accessor function for it) but applications will often need to reference an object's `handle` value whenever making Win32 API calls (which is common) so I decided to make this field public.

For those who are unfamiliar with the Win32 API, a *handle* is an index into an internal Windows data structure that Windows uses to identify various objects. You will treat the handle as an opaque object (that is, the particular value of the handle is meaningless to you; Windows supplies it when you create a Windows' object and you pass it back to Windows when referencing that object). Unless you are creating your own HOWL classes to implement some new object, you'll never have to worry about obtaining a value for this field; HOWL functions automatically fill in the `handle` value when you create a HOWL object.

The `_name` field is simply a string that holds the name of a particular object. HOWL doesn't actually use this field for much; this field is mainly intended for use by the application program. With most HOWL object constructor functions, you supply this string as an argument and HOWL stores a pointer to the string in the `_name` field. After that point, you can use `_name` however you like. In general, however, it should contain the name of the particular object (which, logically, should be a string containing the HLA identifier for the object).

The `wType` field is another read-only private field. This is a 128-bit (lword) array of bits that HOWL uses to determine the type of an object at run time. In the *howl.hhf* header file, there are several constants of the form *`typename`*`_c` for all of the HOWL classes except wBase_t. For example, there is a constant named `wForm_c` associated with the `wForm_t` class. These constants are small integer values in the range 0..127, with a unique integer value for each specific class. When you create an object, the corresponding HOWL constructor procedure will set the corresponding bit position in the `wType` field to a one for that object's class type and all inherited class types. For example, the main form is of type `wForm_t`, so bit position `wForm_t` in `wType` will contain a '1'; however, `wForm_t` is a subclass of `window_t`, which is a subclass of `wContainer_t`, which is a subclass of `wVisual_t`, so the `wType` bit array will also contain '1' bits in positions `window_c, wContainer_c`, and `wVisual_c`. At run time, an application can test an object to see if it is a particular class (or derived from a particular class) by using an instruction like:

```
bt( wContainer_c, object.wType );
```

After executing this instruction, the carry flag will be set if the object is a `wContainer_t` object (or an object whose class is derived from `wContainer_t`).

Note that HOWL makes considerable use of the `wType` field; therefore you should never change the value of this field in your application program after an object has been created and this field is initialized (of course, it doesn't make sense, logically, to change this field's value as the type is fixed at compile time).

The `visible` and `enabled` fields, accessible via their associated accessor functions (read-only, so no mutator functions are available), tell you whether an object is visible (true) or hidden

(false) and whether they are enabled (true) or disabled (false). Note that although there are no mutator functions that directly set these fields, calling the `show` and `hide` methods will set the `visible` field to true or false and calling the `enable` and `disable` methods will similarly affect the enabled field. For those objects that are not visual (e.g., `wFont_t`), the system ignores the values in these fields.

The `visible` and `enabled` fields arguably belong in the `wVisual_t` class, not in `wBase_t` because they only apply to visual objects. However, they were placed in wBase_t because they apply to `wMenuItem_t` objects (that are derived from `wBase_t` but not `wVisual_t`.

The `onHeap` field is true if the current object's storage appears on the heap. Applications should normally ignore this field. An object's destructor will look at this field to determine if it has to free the object's storage when destroying the object. Unless you're writing the destructor function, this field is probably of little interest to you.

The objectID field contains a unique integer number for each object that HOWL creates. Internally, HOWL passes this number along to Windows whenever Windows needs a numeric identifier for objects. Normally, Windows passes this numeric value along with a message to identify controls and other items; HOWL, however, doesn't use this value except for menu items (HOWL tells Windows pass a pointer to the actual HLA object and that's how HOWL identifies things). If you intend to manually process some Windows' messages, this field's value may be of interest to you. However, as this is advanced HOWL programming, we won't consider that use here.

The `nextWidget` field holds a link between HOWL objects held by a `wContainer_t` object. This is a private data field and applications must not mess with its value.

The `parentForm` field points at the `wForm_t` object on which the current object is attached. This field may contain NULL if the current object isn't attached to a particular form (i.e., you've created the object dynamically and you haven't attached it to a form) or if the current object is a `wForm_t` object that is not attached to another form.

The `parentHandle` field contains the handle of the Windows' object of which the current object is a child window. Note that this is not necessarily the same object as the `parentForm` object as you can have multiple containers containing other containers containing the object in question.

The `objectID_g` is a global (hence the "_g") integer used by HOWL to generate a sequence of unique `objectID` values. As a general rule, applications shouldn't mess with this value. In practice, you can increment its value without causing too much harm, but if you ever set its value to something lower than its current value you could create problems.

The `wBase_t` constructor (`create_wBase`) and destructor (`destroy`) functions aren't directly called by application programs.

Applications normally call the `show, hide, enable`, and `disable` methods when processing generic objects. If the underlying object is a visual object, the `hide` and `show` methods will make the object visible on the form or hide it (respectively). If the underlying object is visible and supports enable/disable, then the `enable` and `disable` methods will affect that object accordingly. If the underlying object is not a visual object, then calling these methods will have no effect.

The accessor methods provide access to the private data fields (and `handle`). The mutator functions (set_*) allow you to store a value into some of the private fields; application programs, however, should not be changing the values of the private data fields. Therefore, applications shouldn't call these mutator functions.

The `wBase_t processMessage` method is the message handler of last resort in the HOWL system. It basically calls the default Windows message handler whenever it is called. Application programs should never call `processMessage`. The HOWL main application invokes `processMessage` in response to a message arriving from Windows.

The Manual Solution

Because this is assembly language, you might be interested to know what kind of magic lies behind the `wForm..endwForm` sequence. HOWL is just a library; the HOWL Declarative Language (HDL) generates a class, an object, and a constructor for that class. However, there is nothing stopping you from writing that same code yourself and bypassing the HDL. Most of the tutorials in this series include two versions: one that uses the HDL and one that provides the explicit code that the `wForm..endwForm` statement would generate. These manual versions usually have an 'x' in the name. For example, *001_HelloWorld.hla* is the HDL version, *001x_HelloWorld.hla* is the manual version.

There are a couple of reasons for manually writing the HOWL code rather than using the HDL to create your forms:

- Assembly language programmers are macho and would never consider using an automatic code generator. Okay, this isn't true, but some people prefer to see exactly what machine instructions are being used. These same people probably don't use the HLA high-level control structures, either (good luck with that one when using HOWL; object-oriented code is very messy when using only low-level machine instructions).
- Programs created manually are slightly smaller than those created with HDL. Arguably they are slightly faster, too. But the code that HDL emits is only executed once, so all you're saving is a few microseconds across the execution of your entire program (and a few bytes across the entire application), so this argument is hardly valid.
- Because HDL is implemented with macros, deciphering error messages due to syntax errors in the HDL source can be very messy. So much so, in fact, that it's actually easier to write the code manually in some cases. HDL works great as long as you don't make any mistakes. Syntax errors in HDL can sometimes be a problem to track down (we'll discuss how to do this in a later tutorial when we actually have a complex HDL declaration).

The first step in writing a manual version of a HOWL application is to realize that the wForm..endwForm sequence is really a specialized class declaration. When you have a statement sequence like:

```
wForm( mainAppWindow )
endwForm
```

This translates (roughly) into:

```
type
    mainAppWindow_t:
```

```
        class inherits( wForm_t );

            create_mainAppWindow
            (
                    caption :string;
                    exStyle :dword;
                    style   :dword;
                    parent  :dword;
                    x       :dword;
                    y       :dword;
                    width   :dword;
                    height  :dword;
                    bkgClr  :dword;
                    visible :boolean
            );

        endclass;
        mainAppWindow_p  :pointer to mainAppWindow_t;

static
    vmt( mainAppWindow_t );
    mainAppWindow       :mainAppWindow_t;
    pmainAppWindow      :mainAppWindow_p := &mainAppWindow;
```

Note that the `create_mainAppWindow` procedure is the constructor for the class. This is the procedure that `appStart` calls during program initialization.

The `wForm..endwForm` sequence also constructs the code for the constructor. This is collected into a string that the `mainAppWindow_implementation` macro emits. Here's what that constructor looks like for the Hello World application:

```
procedure mainAppWindow_t.create_mainAppWindow
(
        caption :string;
        exStyle :dword;
        style   :dword;
        parent  :dword;
        x       :dword;
        y       :dword;
        width   :dword;
        height  :dword;
        bkgClr  :dword;
        visible :boolean
);
begin create_mainAppWindow;

    push( eax );
    push( ebx );
    push( ecx );
    push( edx );

   // Standad main form initialization:
   //
   // If a class procedure call (not typical), then allocate storage
   // for this object:

    if( esi = NULL ) then
```

```
        mem.alloc( @size( mainAppWindow_t ));
        mov( eax, esi );
        mov( true, cl );
    else
        mov( this.wBase_private.onHeap, cl );
    endif;

    // Call the wForm_t constructor to do all the default initialization:

    (type wForm_t [esi]).create_wForm
    (
        "mainAppWindow",
        caption,
        exStyle,
        style,
        parent,
        x,
        y,
        width,
        height,
        bkgClr,
        visible
    );

    // Initialize the VMT pointer:

    mov( &mainAppWindow_t._VMT_, this._pVMT_ );

    // Retrieve the onHeap value from above and store it into
    // the onHeap data field:

    mov( cl, this.wBase_private.onHeap );

    this.onCreate();    // Be nice, call this guy (even if empty).
    pop( edx );
    pop( ecx );
    pop( ebx );
    pop( eax );

end create_mainAppWindow;
```

We'll look at these statements group by group. The first section is the standard HLA constructor initialization code:

```
    if( esi = NULL ) then
        mem.alloc( @size( mainAppWindow_t ));
        mov( eax, esi );
        mov( true, cl );
    else
        mov( this.wBase_private.onHeap, cl );
    endif;
```

If ESI is NULL, then this code sequence allocates storage for a new instance of the object on the heap and sets CL to true (this value will be used to initialize the "onHeap" field later in the constructor). ESI will be NULL when you call this constructor as a class constructor, that is, using a calling sequence like this:

```
mainAppWindow_t.create_mainAppWindow( ... );
```

Note that this is a class constructor call (`mainAppWindow_t`), not an object invocation. Normally when you call the constructor, you call it with an object variable like this:

```
mainAppWindow.create_mainAppWindow( ... );
```

(Note the lack of the "_t" at the end of `mainAppWindow`.). This loads ESI with the address of the `mainAppWindow` variable before calling the constructor. Therefore, the code inside the constructor will load the CL register with the current value of the `this.wBase_private.onHeap` variable (which is initialize to zero/false when HOWL loads the program into memory). This is actually the most common use case; the wForm..endwForm automatically creates the `mainAppWindow` variable for you, so it makes sense to always use that variable for the main form.

Generally, you'll only create one instance of the `mainAppWindow_t` class (this is the *singleton* design pattern). Therefore, the inclusion of the object allocation code in the constructor might seem like a waste of time. However, it is quite possible to create multiple instance of a form in your program. This would not typically be done for the main application form, but if you create an application with multiple forms, it's not unreasonable to create multiple instances of those secondary forms. Hence, `wForm..endwForm` will create a generalized constructor to handle all cases. If you're creating the form manually, this is one area where you can optimize things, you can simply delete this code (or better yet, replace it with an assertion that checks to ensure ESI is not NULL).

The next step is to call the wForm_t constructor to do most of the real initialization work. Because the mainAppWindow_t class is derived from `wForm_t`, the parameter lists are identical. Note that you must call the constructor using the syntax `(type wForm_t [esi]).create_wForm` rather than as `wForm_t.create_wForm`. The latter form creates a new object (on the heap) rather than initializing the existing `mainAppWindow_t` object.

```
(type wForm_t [esi]).create_wForm
(
    "mainAppWindow",
    caption,
    exStyle,
    style,
    parent,
    x,
    y,
    width,
    height,
    bkgClr,
    visible
);
```

The only additional argument in this example is the first argument. This is a string with the window's name (not the same as the title string, which is the second parameter). The value of this string is irrelevant for our purposes. I just copied the `mainAppWindow` name here. Other than this first parameter, I've just copied the arguments passed into the `mainAppWindow_t.create_mainAppWindow` constructor directly to the `wForm_t` constructor call.

As with any HLA class constructor, you've got to initialize the object's VMT pointer before leaving the constructor. This is accomplished with the following statement:

```
mov( &mainAppWindow_t._VMT_, this._pVMT_ );
```

This stores the address of the `mainAppWindow_t` VMT (named `mainAppWindow_t._VMT_`, unsurprisingly enough) into the current object's VMT pointer (`this._pVMT_`). Failure to do this will cause the system to crash whenever you call any of the object's methods (which is about to happen).

On last bit of initialization is to set the object's `onHeap` data field. This is accomplished with the following statement:

```
mov( cl, this.wBase_private.onHeap );
```

Note that the code is accessing a private `wBase_t` field here. That's okay. We're writing the constructor for a class derived from `wForm_t` (which is ultimately derived from `wBase_t`). Child classes can legitimately access the private data fields of ancestor classes.

The last thing the constructor does before cleaning up the stack and returning is to call the `onCreate` method. As you saw in the earlier example, this is where various bits of initialization in the main application take place.

Once you get past the constructor for the `mainAppWindow_t` object, the rest of the code looks exactly like the stuff appearing in the HDL version of the Hello World program. You can take a look at the earlier discussion for more details on that code.