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

#2: Buttons

In this second tutorial of this series, we'll take a look at implementing buttons on HOWL forms. Specifically, we'll be looking at Windows' push button user-interface elements.

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 *002_button1.hla, 003_button2.hla, 004_button3.hla,* and *004x_button3.hla*. Though this particular document does not describe *002x_button1.hla* and *003x_button2.hla*, you may also find these files of interest when reading through this tutorial.

Push Buttons:

Push buttons have four major attributes: an (x,y) coordinate position on a form, a size (width and height), a caption, and an *on click event handler*. The event handler is a special kind of HLA procedure (a *widgetProc* in HOWL terminology) that has the following prototype:

```
type
      widgetProc  :procedure( thisPtr:dword; wParam:dword; lParam:dword );
```

HOWL events include things such as text changing, images being painted, obtaining or losing keyboard focus, and clicking on an object. Some objects support a large number of events, some only support only a few events. Whatever the number, most user interface controls ("widgets") support a single main event. For push button objects, that main event is the *onClick* event.

Whenever some widget event occurs, HOWL relays notification of that event to your application by (possibly) calling an event handler you've written and registered with HOWL. For push

buttons, you'd tell HOWL about an onClick handler you've written when you create the button and then HOWL will automatically call that handler when someone presses the button on the form. As noted above, all event handlers in HOWL are written as widgetProcs.

Therefore, to add a button to a form in your application, you need only do two things: declare the button using the HOWL Declarative Language (HDL) and (optionally) write a widgetProc for that button to do something whenever the user presses the button when the application is running.

The *002_button1.hla* program included with the HOWL examples demonstrates how to place a single button on a form. This program adds a single button to the lower right-hand corner of the main form; when the user presses the button at run time, the widgetProc for the button terminates the program (that is, this example adds a "quit" button to the form). Now you can quit the program by pressing a button on the form rather than by having to press the close box on the title bar.

Like most HOWL applications, *002_button1.hla* is an extension of the "Hello World" program from the first tutorial. Indeed, "Hello World" can be thought of as an empty shell application that contains all the scaffolding code needed for every application. To extend the *001_HelloWorld.hla* program there are three changes you need to make:

1. Change the `windowTitle` string at the beginning of the file to a more meaningful name (e.g., "Button Demo #1").
2. Insert the declaration for any new widgets you want between the `wForm` and `endwForm` statements.
3. Add any necessary widgetProcs after the declarations (and prototypes for those widgetProcs before the `wForm` statement).

In order to add a button to the application, you use the `wPushButton` statement in the HDL. This statement uses the following syntax:

```
wPushButton
(
   buttonName,      // Field name in wForm object (Must be an HLA identifier).
   captionString,   // Caption for push button.
   x,               // x position on form for upper left-hand corner of button.
   y,               // y position on form for upper left-hand corner of button.
   w,               // width of button on form.
   h,               // height of button on form.
   onClick          // "on click" event handler (must be NULL or a widgetProc)
)
```

The first argument is the HLA identifier that the HDL will use as this object's name in the class being created by the wForm statement. This name must be unique within the new class' definition (you cannot, for example, have two buttons with the same name). Generally, a nice descriptive name like `quitButton` would be a good choice.

The second argument is a string constant specifying the button's caption text. Windows will display this text within the button when it draws the button on the form.

The third and fourth arguments should be constants that specify the (x,y) position of the button on the form. The upper-left hand corner of the form's *client area* (that is, the part of the window that excludes the title bar and frame) is position (0,0). The x coordinate increases going from left to right and the y coordinate increases going from top to bottom. Note that the size of the form (the `formW` and `formH` constants at the beginning of the source file in the *001_button1.hla*

example code) specifies the size of the entire form, including the title bar and client area. Therefore, the client area (where you can place widgets) is actually somewhat smaller than the values you specify for `formW` and `formH` (which are both 600 for most of the examples in this tutorial series).

The fifth and sixth arguments are constants that specify the width and height of the button on the form. 20 is probably a good minimum for the height (otherwise the caption text may be truncated on the top and bottom). The minimum width will be whatever it takes to completely display the button's caption text.

The last argument is probably the most interesting to us at this point. This is the name of the widgetProc that HOWL will call when someone presses the button. You can supply the value NULL for this argument, in which case HOWL will not call any widgetProc when the button is pressed (and there will be no notification to your program that someone has pressed the button). For push buttons, passing NULL for the onClick handler doesn't make a lot of sense. For other widgets, however, there may be no need for immediate notification when an event occurs, so supplying NULL is not uncommon when declaring other types of widgets.

There is one catch when supplying the name of a widgetProc in the `wPushButton` declaration: HLA requires that you define this symbol before passing it along to `wPushButton`. This is generally handled in a HOWL application by placing a forward or external procedure prototype before the `wForm` statement. In theory, you could actually put the widgetProc code before the `wForm..endwForm` statements but this is not normally done because many widgetProcs will need to reference the class that the `wForm` statement is defining.

Here's the declarations needed to add a "quit" button to a form:

```
// Forward declaration for the onClick widgetProc that we're going to
// call when a button is pressed.

proc onQuit:widgetProc; @forward;



// Here's the main form definition for the app:

wForm( mainAppWindow );


    // Place a quit button in the lower-right-hand corner of the form:

    wPushButton
    (
        button,       // Field name in mainWindow object
        "Quit",       // Caption for push button
        450,          // x position
        525,          // y position
        125,          // width
        25,           // height
        onQuit        // "on click" event handler
    )

endwForm
```

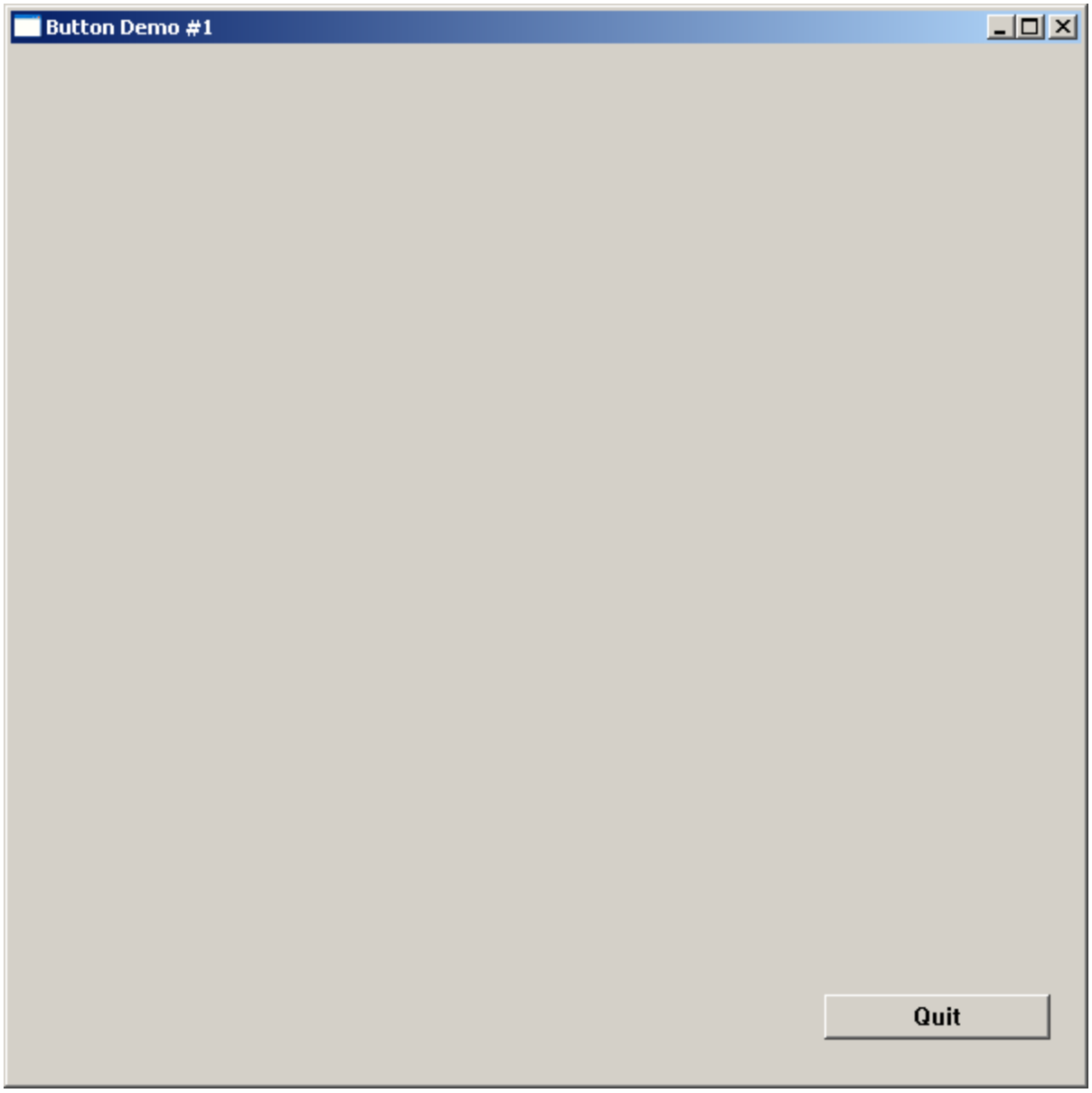
```
// Must invoke the following macro to emit the code generated by
// the wForm macro:

mainAppWindow_implementation();
```

Now all we need to do is supply the `onQuit` widgetProc and we're in business. Here's a straight-forward implementation of `onQuit`:

```
// Here's the onClick event handler for our quit button on the form.
// This handler will simply quit the application:

proc onQuit:widgetProc;
begin onQuit;

    // Quit the app:

    w.PostQuitMessage( 0 );

end onQuit;
```

The Windows' `w.PostQuitMessage` call tells the application to terminate itself. This is roughly equivalent to pressing the close box on the title bar.

If you run this application, you should see the following Window:



A quick comment about widgetProcs is in order here. As noted earlier, the widgetProc type has the following definition:

```
type
        widgetProc  :procedure( thisPtr:dword; wParam:dword; lParam:dword );
```

The examples in this tutorial all use the new-style HLA procedure declarations for widgetProcs, e.g.,

```
proc onQuit:widgetProc;
    ...
```

If you're more comfortable with the original HLA procedure declaration syntax, you'd write the widgetProc thusly:

```
procedure onQuit( thisPtr:dword; wParam:dword; lParam:dword );
begin onQuit;

    // Quit the app:

    w.PostQuitMessage( 0 );

end onQuit;
```

The advantage of the new declaration style is that it's a little safer -- you don't have to worry about getting the spelling (or number) of arguments correct; the advantage of the old syntax is that you get to see the actual parameter declarations when you look at the procedure's code without having to look up the definition of `widgetProc` in the *howl.hhf* header file (on the other hand, you'll write so many widgetProcs when using HOWL that you'll quickly have this parameter list memorized, so this benefit is of dubious value).

In our simple button example, we don't use the values of these arguments, but we should still discuss them because they will be useful when creating other types of widgets.

The `thisPtr` argument is the address of the object associated with this event. When HOWL calls `onQuit` in the current example, `thisPtr` will contain the address of the quit button object (that is, the value held in `mainAppWindow.button`).

The wParam and lParam arguments contain values that Windows passes to HOWL whenever user interaction results in some message being sent to the application (such as when the user presses a button on the form). When the user presses a push button, for example, Windows passes the following values in wParam and lParam:

wParam: the H.O. word contains a special button notification message (that HOWL uses to differentiate the type of button event that has occurred). The L.O. word contains the button's ID (this will be the same value as the `objectID` field from the `wBase_t` class).

lParam: this is the window handle for the button (which should be the same value as the handle field from the `wBase_t` class).

For button onClick widgetProcs, the `wParam` and `lParam` values are redundant. By the time the onClick widgetProc executes HOWL has already decoded the information in `wParam` and we know that it's a button click operation by virtue of the fact that we're executing code in the onClick widgetProc. The lParam value is also redundant because you can obtain this value from the `handle wBase_t` field pointed at by `thisPtr`.

Of course, for the current example, none of these values are important because if someone clicks on the quit button, we're just going to close the application regardless of the values of `thisPtr, wParam,` or `lParam`.

One comment is worth making before moving on: if you have multiple buttons on a form you don't necessarily have to write separate widgetProcs for each button. You can specify the same widgetProc name for all the buttons and then use the `thisPtr` argument to differentiate the buttons. If the buttons are doing very similar things, this might save you from having to write a bunch

of different widgetProcs. On the other hand, if the buttons are doing completely different things (such as quitting the application and saving a file to disk), you're probably better off writing separate widgetProc event handlers for each of the buttons.

The HOWL wVisual_t, wPushButton, wButton, and wClickable_t Classes

Before moving on to the next example, it's worthwhile to quickly discuss the four major classes that HOWL uses to define push buttons. The `wVisual_t` class is an abstract base class that contains information common to all visual objects. The `wClickable_t` class is an abstract base class that handles visual objects that users can click on (such as buttons). The wClickable_t class handles both clicks and double-clicks. The `wButton_t` class is an abstract base class that implements button functionality for the HOWL buttons (push buttons, radio buttons, and check boxes); this is where the "guts" of a push button are found. The `wPushButton_t` class is a concrete implementation of `wButton_t` that HOWL uses to create actual push buttons. The `wButton_t` class has the following definition:

```
wPushButton_t:
    class inherits( wButton_t );

        procedure create_wPushButton
        (
            wpbName     :string;
            caption     :string;
            parent      :dword;
            x           :dword;
            y           :dword;
            width       :dword;
            height      :dword;
            onClick     :widgetProc
        );  external;


    endclass;
```

There really isn't much to explain here. The arguments passed to the constructor (`create_wPushButton`) largely correspond to those in the `wPushButton` HDL declaration. The only difference is the addition of the `parent` argument (which the `wPushButton` declaration automatically fills in for you). The `parent` argument is the handle of the window (form) on which the button is to be placed. If you're placing the button on the form named `mainAppWindow` (the form name that all these tutorials use), then the parent's handle value can be found in `mainAppWindow.handle`.

The real declarations for a button appear in the `wButton_t` class, the immediate parent class of `wPushButton_t`. The `wButton_t` class definition is the following (note that `wButton_t` inherits all the fields of `wClickable_t`):

```
wButton_t:
    class inherits( wClickable_t );

        var
            align( 4 );
```

```
    wButton_private:
        record

            onPaint     :widgetProc;
            onHilite    :widgetProc;
            onUnHilite  :widgetProc;
            onDisable   :widgetProc;
            onSetFocus  :widgetProc;
            onKillFocus :widgetProc;

        endrecord;


    procedure create_wButton
    (
        wbName      :string;
        parent      :dword;
        x           :dword;
        y           :dword;
        width       :dword;
        height      :dword;
        onClick     :widgetProc
    );  external;



    method get_onPaint;     @returns( "eax" );          external;
    method get_onHilite;    @returns( "eax" );          external;
    method get_onUnHilite;  @returns( "eax" );          external;
    method get_onDisable;   @returns( "eax" );          external;
    method get_onSetFocus;  @returns( "eax" );          external;
    method get_onKillFocus; @returns( "eax" );          external;

    method set_onPaint      ( onPaint    :widgetProc ); external;
    method set_onHilite     ( onHilite   :widgetProc ); external;
    method set_onUnHilite   ( onUnHilite :widgetProc ); external;
    method set_onDisable    ( onDisable  :widgetProc ); external;
    method set_onSetFocus   ( onSetFocus :widgetProc ); external;
    method set_onKillFocus  ( onKillFocus:widgetProc ); external;

    method get_text( txt:string );                      external;
    method a_get_text;                                  external;
    method set_text( txt:string );                      external;

    override method processMessage;                     external;


endclass;
```

The private data fields are all widgetProc pointers to functions that HOWL will call based on various events. By default, the create_wButton constructor sets all of these fields to NULL (meaning HOWL will ignore these events).

The onPaint, onHilite, onUnHilite, and onDisable events are laregly obsolete at this time. They are supported for compatibility reasons only (with Windows 3.1). You normally wouldn't respond to these events in a HOWL application so we won't discuss them (or their accessor/mutator functions) any further.

The `onSetFocus` and `onKillFocus` data fields point at widgetProcs that HOWL will call when a button gains or loses the keyboard focus. A button gains focus when the user clicks on it or when the user presses TAB on the keyboard and the selection (focus) switches to the button. A button loses focus when the users tabs off of the key or clicks on some other element on the form. Generally, an application probably doesn't care when a button gets or loses the keyboard focus, but you can trap these events for some special purposes.

As noted earlier, the `wButton_t` constructor initializes the `onSetFocus` and `onKillFocus` fields to NULL. Therefore, HOWL will (by default) ignore any focus events it receives from Windows regarding the button. You can register an event handler for these two events by calling the `set_onSetFocus` and `set_onKillFocus` methods and passing in the address of an appropriate widgetProc to handle the event. When HOWL calls such a widgetProc, the `thisPtr` argument will contain the address of the button object and the `wParam` and `lParam` arguments are basically meaningless.

You will notice that `wButton_t` doesn't have a data field for the onClick event handler. This is because the `wClickable_t` base class provides that functionality and `wButton_t` inherits that functionality from `wClickable_t`. See the discussion of `wClickable_t` in a few paragraphs for more details.

The constructor for `wButton_t` is identical to that of `wPushButton_t`. See the discussion given earlier for `wPushButton_t` for more details regarding the constructor.

The last methods directly defined in this class of interest to us are the `get_text`, `a_get_text`, and `set_text` methods. These methods allow a HOWL to retrieve and change the button's caption (text label) at run time.

The `get_text` method retrieves the current caption and stores it into the string variable you pass as an argument. The string argument must be properly initialized and the string should have sufficient storage to hold all the characters in the button's caption. The `get_text` method will raise an exception if this is not the case.

The `a_get_text` method also retrieves the caption text from a button. It allocates storage for a new string on the heap, copies the label's text to that new string, and then returns a pointer to the new string in the EAX register. It is the caller's responsibility to free this storage (with a call to `str.free`) when it is done using the string data.

The `set_text` method copies the the string you pass as an argument to the button's caption. You are responsible for ensuring that the string actually fits; Windows will clip the character data if it is too long to fit in the space you've set aside for the button.

The `wButton_t` class inherits all the fields of the `wClickable_t` class. The `wClickable_t` class has the following definition:

```
wClickable_t:
    class inherits( wVisual_t );
        var
            align( 4 );
            wClickable_private:
                record

                    onClick     :widgetProc;
```

```
                onDblClick  :widgetProc;

            endrecord;

        procedure create_wClickable
        (
            wcName      :string;
            parent      :dword;
            x           :dword;
            y           :dword;
            width       :dword;
            height      :dword;
            onClick     :widgetProc
        );  external;

        method get_onClick;     @returns( "eax" );       external;
        method get_onDblClick;  @returns( "eax" );       external;

        method set_onClick( onClick  :widgetProc );      external;
        method set_onDblClick( onDblClick :widgetProc ); external;
        method click;                                    external;

    endclass;
```

Note that the `wClickable_t` class inherits all the fields of `wVisual_t`. This means that `wButton_t` and `wPushButton_t` objects also inherit all the fields of `wVisual_t` as they inherit all the fields of `wClickable_t`.

The `wClickable_t` class has two private data fields: `onClick` and `onDblClick`. These point at the widgets that HOWL will call whenever the user clicks on a clickable or double-clickable object. Note that the `onDblClick` field was added to the `wClickable_t` class for convenience. Not all objects that are clickable support double-clicking. If an object does not support double-clicking, then HOWL ignores the value of the `onDlbClick` field.

Again, you will notice that the constructor for this class (`create_wClickable`) has the same argument list as the constructors for `wButton_t` and `wPushButton_t`. About the only thing worth mentioning is that these constructors provide an argument for specifying the `onClick` widgetProc but they do not have an argument for specifying the address of the `onDblClick` widgetProc. This was done this way because most clickable objects will need an on click event handler but few widgets will need an on double click event handler. If you have a button (or other object) that needs to respond to a double click event, then you can manually register an "on double click" widgetProc by calling the `set_onDblClick` method.

There is one other issue concerning the on double click event: whenever someone double clicks on an object Windows will send an on click event to the object on the first click and then an on double click event to the object when the second click comes along. If you have both single and double click event handlers installed on a button (or other wClickable_t object), be aware that HOWL will call both widgetProcs in succession when the user double clicks on the object.

The only other interesting method in this class is the `click` method. If you invoke the `click` method, HOWL will send a message to Windows to tell it to behave as though someone manually clicked on the object. Note that calling the `click` method twice in rapid succession does not simulate a double click.

The `wClickable_t` class inherits all the fields of the `wVisual_t` class. The `wVisual_t` class is the most basic visual class in HOWL (it only inherits fields from `wBase_t`, which is a generic abstract base class). `wVisual_t` has the following definition:

```
wVisual_t:
    class inherits( wBase_t );

        var
            align( 4 );
            wVisual_private:
                record

                    x           :dword;
                    y           :dword;
                    width       :dword;
                    height      :dword;
                    bkgColor    :dword;
                    bkgBrush    :dword;
                    style       :dword;
                    exStyle     :dword;

                endrecord;



        // Constructors/Destructors:

        procedure create_wVisual
        (
            wvName          :string;
            parentHandle    :dword;
            x               :dword;
            y               :dword;
            width           :dword;
            height          :dword
        );  external;


        // Accessor functions:

        method get_x;               @returns( "eax" );  external;
        method get_y;               @returns( "eax" );  external;
        method get_width;           @returns( "eax" );  external;
        method get_height;          @returns( "eax" );  external;
        method get_bkgColor;        @returns( "eax" );  external;
        method get_style;           @returns( "eax" );  external;
        method get_exStyle;         @returns( "eax" );  external;


        method set_x( x:dword );                        external;
        method set_y( y:dword );                        external;
        method set_width( width:dword );                external;
        method set_height( height:dword );              external;
        method set_bkgColor
        (
            bkgColor:dword
        );  external;
```

```
        method move( x:dword; y:dword );                external;
        method resize( width:dword; height:dword );     external;

        method setFocus;                                external;

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

        method onClose;                                 external;
        method onCreate;                                external;

    endclass;
```

The `x, y, width,` and `height` private data fields define a *bounding box* around the object being drawn. For a button object, this defines the rectangle in which Windows will draw the button object. *These are private data fields!* It is very important that you access these fields only via the accessor/mutator functions. Whenever you change one of these values, for example, the mutator function calls notifies Windows that it has to redraw the object to reflect the change in position or size on the form.

The (x,y) coordinate values specify a coordinate that is relative to the form on which the visual object appears, not absolute screen coordinates (the particular form/window is specified by the `parentHandle` argument of the constructor).

The `bkgColor` and `bkgBrush` private data fields specify the color of the background behind the object. The background is that part of the screen within the object's bounding rectangle but not part of the actual object being drawn. For example, a circular object's background would be the four corners of the bounding rectangle up to the edge of the circle within the bounding rectangle. Buttons are a good example of an object that don't have a background because the button comletely fills the bounding rectangle. For buttons (and other objects that don't use a background), these fields are largely ignored.

Note that there is only accessor/mutator functions for the `bkgColor` field, no such functions exist for `bkgBrush`. This is because the `set_bkgColor` method automatically computes the value for the `bkgBrush` field. This is why you should always call the accessor and mutator functions for private data fields: often some field values are computed as a resulting of setting some other field value.

The `style` and `exStyle` fields contain Windows' style information for an object. Some objects use these fields, others don't (and ignore their values). For most objects, the style is set when you create the object (e.g., by calling a class constructor) and the style never changes after that. For this reason, you will note that there are no mutator methods for these fields.

The `set_x, set_y, set_width,` and `set_height` mutators deserve special attention. As mentioned earlier these methods will cause Windows to redraw the object in the new position (or with the new size) whenever you call them. There is one minor problem, however: if you call `set_x` and then call `set_y` to reposition some visual object on the form, Windows will actually wind up drawing the object twice, once for each method call. Though this is very quick, it's still kind of grossly inefficient to redraw the object twice. Worse, drawing it twice may result in some

objectionable flickering. For this reason the wVisual_t class provides two additional methods that will set both the (`x,y`) and (`width,height`) values with a single call (each): `move` and `resize`. A call to `move` allows you to set both the x and y values with a single call (and a single redraw of the object). A call to `resize` lets you set the width and height values with a single call (and a single draw).

The `show` and `hide` methods will make a visual object visible or hidden on the form. These methods override the methods in `wBase_t` (which basically do nothing) and are responsible for telling Windows to show or hide the object. Most of the remaining tutorial examples will demonstrate the use of these two methods.

The `enable` and `disable` methods will (as their name implies) enable and disable a visual object. If an object is disabled, Windows will draw it using grayed text and lines and it will ignore any user-interface requests. For example, if you disable a button Windows will reject any attempt to click on that button. As with the `show` and `hide` methods, these methods override the `wBase_t` methods (which do nothing).

The `onClose` and `onCreate` methods are called by the class constructor and destructor. Application programs should never call these methods. These methods were originally intended for developers who are extending the HOWL class library; though it's likely that these functions will go away in a future release of HOWL.

The wVisual_t destructor (destroy) should never be called by an application. It is automatically called by descendant classes when their destructor is called. Normally, an application only calls the main form's destructor, which is responsible for calling the destructors of all the objects attached to the form.

Another Button Example: Button2

Okay, with a discussion of the various classes associated with buttons out of the way, it's time to look at some additional HOWL examples involving buttons. The first example we'll look at (*002_button2.hla*) demonstrates getting and setting the caption text on a button. This is a very straight-forward extension of the *002_button1* application. In order to demonstrate some additional functionality, we're going to add four new widgetProcs to the program: two of them will simply demonstrate the onSetFocus and onKillFocus events, two of them will be on click handlers (for the new button we add) that will get and set the button's caption text. Here's the declaration for the form:

```
proc onSetFocus1:widgetProc; @forward;
proc onKillFocus1:widgetProc; @forward;
proc onClickChange1:widgetProc; @forward;
proc onClickChange2:widgetProc; @forward;
proc onQuit:widgetProc; @forward;

// Here's the main form definition for the app:

wForm( mainAppWindow );


    wPushButton
    (
```

```
        button1,            // Field name in mainWindow object
        "Press to change",  // Caption for push button
        10,                 // x position
        10,                 // y position
        125,                // width
        25,                 // height
        onClickChange1      // initial "on click" event handler
    )
    // Place a quit button in the lower-right-hand corner of the form:

    wPushButton
    (
        quitButton,   // Field name in mainWindow object
        "Quit",       // Caption for push button
        450,          // x position
        525,          // y position
        125,          // width
        25,           // height
        onQuit        // "on click" event handler
    )

endwForm
```

This particular application is going to have two event handlers that trigger when button1 gets and loses keyboard focus. These widgetProcs are very straight-forward:

```
// The onSetFocus and onKillFocus widgetProcs simply print to the console
// what has happened.

proc onSetFocus1:widgetProc;
begin onSetFocus1;

    stdout.put( "Set focus to button 1" nl );

end onSetFocus1;


proc onKillFocus1:widgetProc;
begin onKillFocus1;

    stdout.put( "Shifted focus from button 1" nl );

end onKillFocus1;
```

These two methods will simply print an informative string to the console window whenever button1 receives or loses the keyboard focus. Note that you shouldn't compile this code with the HLA `-w` command-line option or you should always run the application from a command-line so that you can see the data these methods send to the standard output device.

Because the wPushButton declaration doesn't let you set the `onSetFocus` and `onKillFocus` widgetProc pointers, you're going to have to explicitly call the `wButton_t` mutator functions to accomplish this. The only question is where do you call these methods from? You can't place the code to do this in the `wForm..endwForm` sequence: remember, that code is the declaration of a class, you can't arbitrarily place executable code there. The best solution is to place initialization code (such as setting up event handler addresses) in the `mainAppWindow.onCreate` method (that

is part of the skeleton program we inherited from Hello World. HOWL calls `mainAppWindow.onCreate` after it has created the form object and immediately before returning to its caller. You could also place the code in the appStart procedure, immediately after the call to the `mainAppWindow` constructor call, but the standard HOWL convention is to place the code in the `onCreate` method. Here's what the modified `onCreate` method looks like:

```
// 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;

    // Lets set up the button1's onSetFocus and onKillFocus event handlers:

    mov( this.button1, esi );
    (type wPushButton_t [esi]).set_onSetFocus( &onSetFocus1 );
    (type wPushButton_t [esi]).set_onKillFocus( &onKillFocus1 );

end onCreate;
```

This function loads the pointer to the `mainAppWindow.button1` object into ESI (this is the pointer to the button object we want to attach the event handlers to) and then calls the `set_onSetFocus` and `set_onKillFocus` methods to set those data fields to point at the `onSetFocus1` and `onSetFocus2` widgetProcs given earlier.

Note the use of type coercion in this example. Remember that this `onCreate` method is a member of the `mainAppWindow_t class`, not a member of the `wButton_t` class. Within this method "this" refers to an object of type `mainAppWindow_t`. After loading this.button1 into ESI, this code has to coerce ESI to point at a `wPushButton_t` object in order to correctly access the methods that set the focus widgetProcs.

Next, let's take a look at the `onClickChange1` widgetProc that the system (initially) calls when the user presses `button1` on the form:

```
proc onClickChange1:widgetProc;
var
    curCaption    :string;
    curCapBuf     :char[256];

begin onClickChange1;

    str.init( curCapBuf, @size( curCapBuf ));
    mov( eax, curCaption );

    mov( thisPtr, esi );    // ESI already contains this, but just in case...

    // Print the current caption to the console window:

    (type wPushButton_t [esi]).get_text( curCaption );
    stdout.put( "Current caption1: ", curCaption, nl );


    // Change the caption:

    (type wPushButton_t [esi]).set_text( "Restore original" );
```

```
    // Point the onClick handler at onClickChange2:

    (type wPushButton_t [esi]).set_onClick( &onClickChange2 );


    // Print the new caption to the console window:

    (type wPushButton_t [esi]).a_get_text();
    stdout.put( "New caption1: ", (type string eax), nl nl );
    str.free( eax );

end onClickChange1;
```

The first thing to remember about a widgetProc is that it is not a class method. Therefore, you cannot use the HLA "this" reserved word inside the procedure to access fields of the object associated with the widgetProc. Also note that you do not have to save any register values inside a widgetProc -- HOWL automatically preserves the important registers for you before it calls your widgetProc. In theory, ESI will contain a pointer to the object that invoked the widgetProc (that is, the value of `thisPtr`); however, it's much safer to load ESI with the value of `thisPtr` upon entry into the widgetProc, just in case some future HOWL code doesn't do this for you.

After setting up ESI with the value of `thisPtr`, this widgetProc demonstrates reading the button's caption text into a local string variable (and then it prints this string to the console). After fetching the string, this code then changes the string to "Restore Original". Next, the code changes the `onClick` widgetProc address to point at the `onClickChange2` procedure. That means that the next time the user presses the button HOWL will call `onClickChange2` rather than `onClickChange1`. This code demonstrates a simple way to implement state machines in a HOWL application -- by simply changing the event handler addresses in response to events.

The last thing that `onClickChange1` does is demonstrate the `a_get_text` method by call that method to fetch the new string we've just assigned to the button and printing it on the console. Like all good functions that call `a_get_text`, this procedure calls `str.free` to free the storage associated with the string when it's done using it.

The `onClickChange2` handler is very similar to `onClickChange1` and takes the following form:

```
proc onClickChange2:widgetProc;
var
    curCaption:string;
    curCapBuf:char[256];

begin onClickChange2;

    str.init( curCapBuf, @size( curCapBuf ));
    mov( eax, curCaption );

    mov( thisPtr, esi );// ESI already contains this, but just in case...

    // Print the current caption to the console window:

    (type wPushButton_t [esi]).get_text( curCaption );
    stdout.put( "Current caption2: ", curCaption, nl );
```

```
    // Change the caption:

    (type wPushButton_t [esi]).set_text( "Press to change" );

    // Point the onClick handler at onClickChange1:

    (type wPushButton_t [esi]).set_onClick( &onClickChange1 );


    // Print the new caption to the console window:

    (type wPushButton_t [esi]).a_get_text();
    stdout.put( "New caption2: ", (type string eax), nl nl );
    str.free( eax );

end onClickChange2;
```

Note that this code changes the caption back to its original text and resets the `onClick` widgetProc pointer to point back at `onClickChange1`.

If you run this program, you can watch the text toggle between the two strings “Press to change” and “Restore original”.

Here is the complete program (last time I’ll do this to you):

```
// button2-
//
//  This program demonstrates changing a button's caption and other attributes
// under program control,

program button2;
#linker( "comdlg32.lib" )
#linker( "comctl32.lib" )

?compileAll     := true;

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

#includeOnce( "stdlib.hhf" )

#includeOnce( "howl.hhf" )


const
    applicationName := "Button #2";
    formX           := w.CW_USEDEFAULT; // Let Windows position this guy
    formY           := w.CW_USEDEFAULT;
    formW           := 600;
    formH           := 600;



// Forward declarations for the onClick widgetProcs that we're going to
// call when a button is pressed.

proc onSetFocus1:widgetProc; @forward;
proc onKillFocus1:widgetProc; @forward;
```

```
proc onClickChange1:widgetProc; @forward;
proc onClickChange2:widgetProc; @forward;
proc onQuit:widgetProc; @forward;



// Here's the main form definition for the app:

wForm( mainAppWindow );


    wPushButton
    (
        button1,                // Field name in mainWindow object
        "Press to change",  // Caption for push button
        10,                 // x position
        10,                 // y position
        125,                // width
        25,                 // height
        onClickChange1      // initial "on click" event handler
    )
    // Place a quit button in the lower-right-hand corner of the form:

    wPushButton
    (
        quitButton,         // Field name in mainWindow object
        "Quit",             // Caption for push button
        450,                // x position
        525,                // y position
        125,                // width
        25,                 // height
        onQuit              // "on click" event handler
    )

endwForm


// Must invoke the following macro to emit the code generated by
// the wForm macro:

mainAppWindow_implementation();



// The onSetFocus and onKillFocus widgetProcs simply print to the console
// what has happened.

proc onSetFocus1:widgetProc;
begin onSetFocus1;

    stdout.put( "Set focus to button 1" nl );

end onSetFocus1;


proc onKillFocus1:widgetProc;
begin onKillFocus1;

    stdout.put( "Shifted focus from button 1" nl );
```

```
end onKillFocus1;


// Here's 1 of 2 onClick handlers for button1. This widgetProc
// changes the caption to "Restore caption" and sets the
// onClick pointer to point at the second onClick handler.


proc onClickChange1:widgetProc;
var
    curCaption  :string;
    curCapBuf   :char[256];

begin onClickChange1;

    str.init( curCapBuf, @size( curCapBuf ));
    mov( eax, curCaption );

    mov( thisPtr, esi );        // ESI already contains this, but just in case...

    // Print the current caption to the console window:

    (type wPushButton_t [esi]).get_text( curCaption );
    stdout.put( "Current caption1: ", curCaption, nl );


    // Change the caption:

    (type wPushButton_t [esi]).set_text( "Restore original" );

    // Point the onClick handler at onClickChange2:

    (type wPushButton_t [esi]).set_onClick( &onClickChange2 );


    // Print the new caption to the console window:

    (type wPushButton_t [esi]).a_get_text();
    stdout.put( "New caption1: ", (type string eax), nl nl );
    str.free( eax );

end onClickChange1;



// Here's 2 of 2 onClick handlers for button1. This widgetProc
// changes the caption back to "Restore caption" and sets the
// onClick pointer to point at the first onClick handler.


proc onClickChange2:widgetProc;
var
    curCaption  :string;
    curCapBuf   :char[256];

begin onClickChange2;

    str.init( curCapBuf, @size( curCapBuf ));
```

```
    mov( eax, curCaption );

    mov( thisPtr, esi );        // ESI already contains this, but just in case...

    // Print the current caption to the console window:

    (type wPushButton_t [esi]).get_text( curCaption );
    stdout.put( "Current caption2: ", curCaption, nl );


    // Change the caption:

    (type wPushButton_t [esi]).set_text( "Press to change" );

    // Point the onClick handler at onClickChange1:

    (type wPushButton_t [esi]).set_onClick( &onClickChange1 );


    // Print the new caption to the console window:

    (type wPushButton_t [esi]).a_get_text();
    stdout.put( "New caption2: ", (type string eax), nl nl );
    str.free( eax );

end onClickChange2;



// Here's the onClick event handler for our quit button on the form.
// This handler will simply quit the application:

proc onQuit:widgetProc;
begin onQuit;

    // Quit the app:

    w.PostQuitMessage( 0 );

end onQuit;



// 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;

    // Lets set up the button1's onSetFocus and onKillFocus event handlers:

    mov( this.button1, esi );
    (type wPushButton_t [esi]).set_onSetFocus( &onSetFocus1 );
    (type wPushButton_t [esi]).set_onKillFocus( &onKillFocus1 );

end onCreate;


////////////////////////////////////////////////////////////////////////////
```

```
//
//
// 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.  Notice the
// override declaration for onClose in the wForm declaration given earlier.
// Without that, mainAppWindow_t would default to using the wVisual_t.onClose
// method (which does nothing).

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,                  // Form x-coordinate
        formY,                  // Form y-coordinate
        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
// call the HowlMainApp procedure.

begin button2;

    // 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 );

    // Start the HOWL Framework Main Program:

    HowlMainApp();
```

```
    // Delete the brush we created earlier:

    w.DeleteObject( howl.bkgBrush_g );

end button2;
```

Button3: Demonstrating Lots of Button Methods

The *button2* example demonstrated how to change the caption on a push button. In this example we're going to expand our examination of the various methods you can call to affect a button. We'll look at making buttons invisible and visible, enabling and disabling buttons, moving and resizing buttons, clicking buttons under program control, and activating the onDblClick, onSetFocus, and onKillFocus events.

The *button3* program uses the *button2* program as its base code and extends that program by adding five new buttons to the form. Here is the initial code just prior to the HDL declarations in the source file:

```
// button3-
//
//  This program expands on button2 by demonstrating multiple buttons,
// simulated button clicks, double clicks, showing and hiding buttons,
// enabling and disabling buttons, moving buttons, and resizing buttons.

program button3;
#linker( "comdlg32.lib" )
#linker( "comctl32.lib" )

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

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

const
    applicationName := "Button Demo #3";
    formX           := w.CW_USEDEFAULT; // Let Windows position this guy
    formY           := w.CW_USEDEFAULT;
    formW           := 600;
    formH           := 600;



// Forward declarations for the onClick widgetProcs that we're going to
// call when a button is pressed.

proc onSetFocus1            :widgetProc; @forward;
proc onKillFocus1           :widgetProc; @forward;
proc onClickChange1         :widgetProc; @forward;
proc onClickChange2         :widgetProc; @forward;
proc hideShowButton         :widgetProc; @forward;
proc enableDisableButton    :widgetProc; @forward;
proc moveButton             :widgetProc; @forward;
proc resizeButton           :widgetProc; @forward;
```

```
proc onDblClick             :widgetProc; @forward;
proc onQuit                 :widgetProc; @forward;
```

Most of this code is straight out of the *button2* example. The only thing really different here is the addition of several new widgetProc prototypes that the new buttons on the form will call. Now, let's take a look at the first part of the HDL declaration section:

```
// Here's the main form definition for the app:

wForm( mainAppWindow );

    var
        showState   :boolean;
        b1Enabled   :boolean;
        align(4);
```

Here's a big difference from the earlier examples in this tutorial: variable declarations appearing within a `wForm..endwForm` statement. Technically, these could have been global variables, but they were stuck within the `wForm..endwForm` sequence to demonstrate (and reinforce) that the `wForm..endwForm` sequence is a declaration, not code. Any statement that is legal within a `class..endclass` declaration is legal within a `wForm..endwForm` sequence. The `var` declarations here add two boolean data fields to the `mainAppWindow` class we are creating. Throughout the program we can refer to these data fields using `mainAppWindow.showState` and `mainAppWindow.blEnabled`. Okay, let's take a look at the button declarations (that constitute the remainder of the `wForm..endwForm` declaration):

```
    wPushButton
    (
        button1,                // Field name in mainWindow object
        "Button #1",            // Caption for push button
        10,                     // x position
        10,                     // y position
        125,                    // width
        25,                     // height
        onClickChange1          // initial "on click" event handler
    )


    wPushButton
    (
        button2,                // Field name in mainWindow object
        "Hide button 1",        // Caption for push button
        175,                    // x position
        10,                     // y position
        125,                    // width
        25,                     // height
        hideShowButton          // initial "on click" event handler
    )


    wPushButton
    (
        button3,                // Field name in mainWindow object
        "Disable button 1",     // Caption for push button
```

```
        175,                    // x position
        40,                     // y position
        125,                    // width
        25,                     // height
        enableDisableButton     // initial "on click" event handler
    )


    wPushButton
    (
        button4,                // Field name in mainWindow object
        "Move button 1",        // Caption for push button
        175,                    // x position
        70,                     // y position
        125,                    // width
        25,                     // height
        moveButton              // initial "on click" event handler
    )


    wPushButton
    (
        button5,                // Field name in mainWindow object
        "Resize button 1",      // Caption for push button
        175,                    // x position
        100,                    // y position
        125,                    // width
        25,                     // height
        resizeButton            // initial "on click" event handler
    )


    wPushButton
    (
        button6,                // Field name in mainWindow object
        "DblClick to Click",    // Caption for push button
        175,                    // x position
        130,                    // y position
        125,                    // width
        25,                     // height
        NULL                    // no single click handler
    )


    // Place a quit button in the lower-right-hand corner of the form:

    wPushButton
    (
        quitButton,             // Field name in mainWindow object
        "Quit",                 // Caption for push button
        450,                    // x position
        525,                    // y position
        125,                    // width
        25,                     // height
        onQuit                  // "on click" event handler
    )

endwForm
```

```
// Must invoke the following macro to emit the code generated by
// the wForm macro:

mainAppWindow_implementation();
```

There is nothing special here. Just more of the button declarations you've seen in earlier examples. The new stuff, of course, is in the widgetProcs for the new buttons.

The first new widgetProc is associated with *button6*. If you look at the declaration for *button6* in the `wForm..endwForm` sequence, you notice that the `onClick` event handler is NULL. The *button6* object is going to demonstrate double-clicking on a button and a program must manually set the `onDblClick` event handler; you'll see the code that does this later on in the source file. In the meantime, let's look at this widgetProc that handles double-clicking on *button6*:

```
// The onDblClick widget proc will handle a double click on button6
// and simulate a single click on button 1.

proc onDblClick:widgetProc;
begin onDblClick;

    mov( mainAppWindow.button1, esi );
    (type wPushButton_t [esi]).click();

end onDblClick;
```

Whenever the system calls this widgetProc, it loads the `button1` object pointer into ESI and then calls the `click` method for `button1`. Calling the `click` method simulates a button click on the associated object; therefore, double-clicking on *button6* will cause a single-click operation on *button1*.

The next widgetProc in the source file is `resizeButton`. HOWL will call this widgetProc when the user presses *button5*. Here is the code for the `resizeButton` procedure:

```
// The resizeButton widget proc will resize button1 between widths 125 and 150.

proc resizeButton:widgetProc;
begin resizeButton;

    mov( mainAppWindow.button1, esi );
    (type wPushButton_t [esi]).get_width();
    if( eax = 125 ) then

        stdout.put( "Resizing button to width 150" nl );
        (type wPushButton_t [esi]).set_width( 150 );

    else

        stdout.put( "Resizing button to width 125" nl );
        (type wPushButton_t [esi]).set_width( 125 );

    endif;

end resizeButton;
```

The `resizeButton` widgetProc demonstrates two `wButton` method calls: `get_width` and `set_width`. This function calls `get_width` to determine the current width of *button1* and then sets *button1*'s size to 150 pixels if the current width is 125, it sets the width to 125 pixels if the current width is not 125 (presumably, it will be 150 if it is not 125). This code also displays the new width on the console window.

The next widgetProc in the source file is the `moveButton` procedure. This procedure is quite similar to `resizeButton` except it changes the y-coordinate value of *button1*'s position on the screen rather than the width of the button. This procedure alternates the button's position between y-coordinates 10 and 40:

```
// The moveButton widget proc will move button1 between y positions 10 and 40.

proc moveButton:widgetProc;
begin moveButton;

    mov( mainAppWindow.button1, esi );
    (type wPushButton_t [esi]).get_y();
    if( eax = 10 ) then

        stdout.put( "Moving button to y-position 40" nl );
        (type wPushButton_t [esi]).set_y( 40 );

    else

        stdout.put( "Moving button to y-position 10" nl );
        (type wPushButton_t [esi]).set_y( 10 );

    endif;

end moveButton;
```

The next widgetProc in the source file is the `enableDisableButton` procedure. This widgetProc alternately enables or disables *button1* on the form:

```
// The enableDisableButton widget proc will enable and disable button1.

proc enableDisableButton:widgetProc;
begin enableDisableButton;

    mov( thisPtr, esi );
    if( mainAppWindow.b1Enabled ) then

        (type wPushButton_t [esi]).set_text( "Enable button 1" );
        mov( false, mainAppWindow.b1Enabled );
        stdout.put( "Disabling button 1" nl );
        mov( mainAppWindow.button1, esi );
        (type wPushButton_t [esi]).disable();

    else

        (type wPushButton_t [esi]).set_text( "Disable button 1" );
        mov( true, mainAppWindow.b1Enabled );
```

```
        stdout.put( "Enabling button 1" nl );
        mov( mainAppWindow.button1, esi );
        (type wPushButton_t [esi]).enable();

    endif;

end enableDisableButton;
```

The interesting thing to note about `enableDisableButton` is how it accesses the `mainAppWindow.b1Enabled` variable to determine whether the button is currently enabled or disabled. This procedure calls *button1*'s `enable` or `disable` methods in order to toggle the current state. It also sets the caption on *button3* (`thisPtr` points at `button3`) to reflect the operation that pressing the button will perform.

The next widgetProc is the `hideShowButton` procedure. This procedure checks the showState class variable to determine whether it should make *button1* visible or invisible. It calls *button1*'s `show` method to make it visible and *button1*'s `hide` method to make it invisible. This method also displays the new status to the console window and updates *button2*'s caption to reflect the operation that pressing this button will perform:

```
// The hideShowButton widget proc will hide and show button1.

proc hideShowButton:widgetProc;
begin hideShowButton;

    mov( thisPtr, esi );
    if( mainAppWindow.showState ) then

        (type wPushButton_t [esi]).set_text( "Hide button 1" );
        mov( false, mainAppWindow.showState );
        stdout.put( "Showing button 1" nl );
        mov( mainAppWindow.button1, esi );
        (type wPushButton_t [esi]).show();

    else

        (type wPushButton_t [esi]).set_text( "Show button 1" );
        mov( true, mainAppWindow.showState );
        stdout.put( "Hiding button 1" nl );
        mov( mainAppWindow.button1, esi );
        (type wPushButton_t [esi]).hide();

    endif;

end hideShowButton;
```

The next two widgetProcs in the source file, `onSetFocus1` and `onKillFocus1`, are fairly trivial. All they do is print a string to the console window telling the user what has happened. These widgetProcs exist mainly to demonstrate how you activate the `onSetFocus` and `onKillFocus` events. Here is the code for these two procedures:

```
// The onSetFocus and onKillFocus widgetProcs simply print to the console
// what has happened.
```

```
proc onSetFocus1:widgetProc;
begin onSetFocus1;

    stdout.put( "Set focus to button 1" nl );

end onSetFocus1;


proc onKillFocus1:widgetProc;
begin onKillFocus1;

    stdout.put( "Shifted focus from button 1" nl );

end onKillFocus1;
```

The next three widgetProcs, `onClickChange1`, `onClickChange2`, and `onQuit` are copied straight from the previous example in this tutorial, so there is no need to repeat their code and description here.

The last piece of code of interest to us in this example is the `mainAppWindow_t.onCreate` method. This method is where the initialization of the new class data field variables takes place and where the program sets up the `onDblClick`, `onKillFocus`, and `onSetFocus` event handlers:

```
method mainAppWindow_t.onCreate;
var
    thisPtr :dword;

begin onCreate;

    mov( esi, thisPtr );

    // Initialize the showState and enableDisableButton data fields:

    mov( false, this.showState );
    mov( true, this.b1Enabled );

    // Lets set up the button1's onSetFocus and onKillFocus event handlers:

    mov( this.button1, esi );
    (type wPushButton_t [esi]).set_onSetFocus( &onSetFocus1 );
    (type wPushButton_t [esi]).set_onKillFocus( &onKillFocus1 );

    // Set up button6's onDblClick handler:

    mov( thisPtr, esi );
    mov( this.button6, esi );
    (type wPushButton_t [esi]).set_onDblClick( &onDblClick );

end onCreate;
```

Note that because this is an actual method in `mainAppWindow_t` class we can use the this pointer to access fields of the `mainAppWindow` object.

Okay, here's the full source code to the `004_button3.hla` file:

```
// button3-
//
//  This program expands on button2 by demonstrating multiple buttons,
// simulated button clicks, double clicks, showing and hiding buttons,
// enabling and disabling buttons, moving buttons, and resizing buttons.

program button3;
#linker( "comdlg32.lib" )
#linker( "comctl32.lib" )

?compileAll     := true;

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

#includeOnce( "stdlib.hhf" )

#includeOnce( "howl.hhf" )



const
    applicationName := "Button Demo #3";
    formX           := w.CW_USEDEFAULT; // Let Windows position this guy
    formY           := w.CW_USEDEFAULT;
    formW           := 600;
    formH           := 600;



// Forward declarations for the onClick widgetProcs that we're going to
// call when a button is pressed.

proc onSetFocus1            :widgetProc; @forward;
proc onKillFocus1           :widgetProc; @forward;
proc onClickChange1         :widgetProc; @forward;
proc onClickChange2         :widgetProc; @forward;
proc hideShowButton         :widgetProc; @forward;
proc enableDisableButton    :widgetProc; @forward;
proc moveButton             :widgetProc; @forward;
proc resizeButton           :widgetProc; @forward;
proc onDblClick             :widgetProc; @forward;
proc onQuit                 :widgetProc; @forward;



// Here's the main form definition for the app:

wForm( mainAppWindow );

    var
        showState   :boolean;
        b1Enabled   :boolean;
        align(4);

    wPushButton
    (
        button1,                // Field name in mainWindow object
        "Button #1",            // Caption for push button
```

```
    10,                     // x position
    10,                     // y position
    125,                    // width
    25,                     // height
    onClickChange1          // initial "on click" event handler
)


wPushButton
(
    button2,                // Field name in mainWindow object
    "Hide button 1",        // Caption for push button
    175,                    // x position
    10,                     // y position
    125,                    // width
    25,                     // height
    hideShowButton          // initial "on click" event handler
)


wPushButton
(
    button3,                // Field name in mainWindow object
    "Disable button 1",     // Caption for push button
    175,                    // x position
    40,                     // y position
    125,                    // width
    25,                     // height
    enableDisableButton     // initial "on click" event handler
)


wPushButton
(
    button4,                // Field name in mainWindow object
    "Move button 1",        // Caption for push button
    175,                    // x position
    70,                     // y position
    125,                    // width
    25,                     // height
    moveButton              // initial "on click" event handler
)


wPushButton
(
    button5,                // Field name in mainWindow object
    "Resize button 1",      // Caption for push button
    175,                    // x position
    100,                    // y position
    125,                    // width
    25,                     // height
    resizeButton            // initial "on click" event handler
)


wPushButton
(
    button6,                // Field name in mainWindow object
```

```
        "DblClick to Click",    // Caption for push button
        175,                    // x position
        130,                    // y position
        125,                    // width
        25,                     // height
        NULL                    // no single click handler
    )


    // Place a quit button in the lower-right-hand corner of the form:

    wPushButton
    (
        quitButton,             // Field name in mainWindow object
        "Quit",                 // Caption for push button
        450,                    // x position
        525,                    // y position
        125,                    // width
        25,                     // height
        onQuit                  // "on click" event handler
    )

endwForm


// Must invoke the following macro to emit the code generated by
// the wForm macro:

mainAppWindow_implementation();







// The onDblClick widget proc will handle a double click on button6
// and simulate a single click on button 1.

proc onDblClick:widgetProc;
begin onDblClick;

    mov( mainAppWindow.button1, esi );
    (type wPushButton_t [esi]).click();

end onDblClick;



// The resizeButton widget proc will resize button1 between widths 125 and 150.

proc resizeButton:widgetProc;
begin resizeButton;

    mov( mainAppWindow.button1, esi );
    (type wPushButton_t [esi]).get_width();
    if( eax = 125 ) then

        stdout.put( "Resizing button to width 150" nl );
```

```
        (type wPushButton_t [esi]).set_width( 150 );

    else

        stdout.put( "Resizing button to width 125" nl );
        (type wPushButton_t [esi]).set_width( 125 );

    endif;

end resizeButton;



// The moveButton widget proc will move button1 between y positions 10 and 40.

proc moveButton:widgetProc;
begin moveButton;

    mov( mainAppWindow.button1, esi );
    (type wPushButton_t [esi]).get_y();
    if( eax = 10 ) then

        stdout.put( "Moving button to y-position 40" nl );
        (type wPushButton_t [esi]).set_y( 40 );

    else

        stdout.put( "Moving button to y-position 10" nl );
        (type wPushButton_t [esi]).set_y( 10 );

    endif;

end moveButton;



// The enableDisableButton widget proc will enable and disable button1.

proc enableDisableButton:widgetProc;
begin enableDisableButton;

    mov( thisPtr, esi );
    if( mainAppWindow.b1Enabled ) then

        (type wPushButton_t [esi]).set_text( "Enable button 1" );
        mov( false, mainAppWindow.b1Enabled );
        stdout.put( "Disabling button 1" nl );
        mov( mainAppWindow.button1, esi );
        (type wPushButton_t [esi]).disable();

    else

        (type wPushButton_t [esi]).set_text( "Disable button 1" );
        mov( true, mainAppWindow.b1Enabled );
        stdout.put( "Enabling button 1" nl );
        mov( mainAppWindow.button1, esi );
        (type wPushButton_t [esi]).enable();

    endif;
```

```
end enableDisableButton;



// The hideShowButton widget proc will hide and show button1.

proc hideShowButton:widgetProc;
begin hideShowButton;

    mov( thisPtr, esi );
    if( mainAppWindow.showState ) then

        (type wPushButton_t [esi]).set_text( "Hide button 1" );
        mov( false, mainAppWindow.showState );
        stdout.put( "Showing button 1" nl );
        mov( mainAppWindow.button1, esi );
        (type wPushButton_t [esi]).show();

    else

        (type wPushButton_t [esi]).set_text( "Show button 1" );
        mov( true, mainAppWindow.showState );
        stdout.put( "Hiding button 1" nl );
        mov( mainAppWindow.button1, esi );
        (type wPushButton_t [esi]).hide();

    endif;

end hideShowButton;



// The onSetFocus and onKillFocus widgetProcs simply print to the console
// what has happened.

proc onSetFocus1:widgetProc;
begin onSetFocus1;

    stdout.put( "Set focus to button 1" nl );

end onSetFocus1;


proc onKillFocus1:widgetProc;
begin onKillFocus1;

    stdout.put( "Shifted focus from button 1" nl );

end onKillFocus1;


// Here's 1 of 2 onClick handlers for button1. This widgetProc
// changes the caption to "Restore caption" and sets the
// onClick pointer to point at the second onClick handler.


proc onClickChange1:widgetProc;
var
```

```
    curCaption  :string;
    curCapBuf   :char[256];

begin onClickChange1;

    str.init( curCapBuf, @size( curCapBuf ));
    mov( eax, curCaption );

    mov( thisPtr, esi );        // ESI already contains this, but just in case...

    // Print the current caption to the console window:

    (type wPushButton_t [esi]).get_text( curCaption );
    stdout.put( "Current caption1: ", curCaption, nl );


    // Change the caption:

    (type wPushButton_t [esi]).set_text( "Restore Button #1" );

    // Point the onClick handler at onClickChange2:

    (type wPushButton_t [esi]).set_onClick( &onClickChange2 );


    // Print the new caption to the console window:

    (type wPushButton_t [esi]).a_get_text();
    stdout.put( "New caption1: ", (type string eax), nl nl );
    str.free( eax );

end onClickChange1;



// Here's 2 of 2 onClick handlers for button1. This widgetProc
// changes the caption back to "Restore caption" and sets the
// onClick pointer to point at the first onClick handler.


proc onClickChange2:widgetProc;
var
    curCaption  :string;
    curCapBuf   :char[256];

begin onClickChange2;

    str.init( curCapBuf, @size( curCapBuf ));
    mov( eax, curCaption );

    mov( thisPtr, esi );        // ESI already contains this, but just in case...

    // Print the current caption to the console window:

    (type wPushButton_t [esi]).get_text( curCaption );
    stdout.put( "Current caption2: ", curCaption, nl );


    // Change the caption:
```

```
    (type wPushButton_t [esi]).set_text( "Button #1" );

    // Point the onClick handler at onClickChange1:

    (type wPushButton_t [esi]).set_onClick( &onClickChange1 );


    // Print the new caption to the console window:

    (type wPushButton_t [esi]).a_get_text();
    stdout.put( "New caption2: ", (type string eax), nl nl );
    str.free( eax );

end onClickChange2;



// Here's the onClick event handler for our quit button on the form.
// This handler will simply quit the application:

proc onQuit:widgetProc;
begin onQuit;

    // Quit the app:

    w.PostQuitMessage( 0 );

end onQuit;




// We'll use the main application form's onCreate method to initialize
// the various buttons on the form.
//
// This could be done in appStart, but better to leave appStart mainly
// as boilerplate code. Also, putting this code here allows us to use
// "this" to access the mainAppWindow fields (a minor convenience).

method mainAppWindow_t.onCreate;
var
    thisPtr :dword;

begin onCreate;

    mov( esi, thisPtr );

    // Initialize the showState and enableDisableButton data fields:

    mov( false, this.showState );
    mov( true, this.b1Enabled );

    // Lets set up the button1's onSetFocus and onKillFocus event handlers:

    mov( this.button1, esi );
    (type wPushButton_t [esi]).set_onSetFocus( &onSetFocus1 );
    (type wPushButton_t [esi]).set_onKillFocus( &onKillFocus1 );
```

```
    // Set up button6's onDblClick handler:

    mov( thisPtr, esi );
    mov( this.button6, esi );
    (type wPushButton_t [esi]).set_onDblClick( &onDblClick );

end onCreate;



////////////////////////////////////////////////////////////////////////////
//
//
// 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.  Notice the
// override declaration for onClose in the wForm declaration given earlier.
// Without that, mainAppWindow_t would default to using the wVisual_t.onClose
// method (which does nothing).

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,                  // Form x-coordinate
        formY,                  // Form y-coordinate
        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 button3;

   // 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 );

    // Start the HOWL Framework Main Program:

    HowlMainApp();

    // Delete the brush we created earlier:

    w.DeleteObject( howl.bkgBrush_g );

end button3;
```

The Manual Solution

Once again, it's time to take a look at how to manually implement the HDL code directly in HLA. We're only going to see how this is done in a couple of chapters; however, the last chapter didn't really demonstrate much, so this chapter will show you the general mechanisms behind creating a HOWL application without using the HDL. Because the latter examples in this tutorial have simply been extensions of the previous examples, we'll only consider the *004x_button3.hla* source file in this section.

Everything in *004_button3.hla* and *004x_button3.hla* are largely identical up to the wForm statement. At that point, the two source files diverge for a while; we'll discuss the parts of the files that are actually different.

Remember, the wForm..endwForm statement in *004_button3.hla* is a class declaration. Therefore, the `wForm( mainAppWindow )` clause is going to translate into an HLA class declaration:

```
type

    // Create a new class for our main application window.
    // All application forms must be derived from wForm_t:

    mainAppWindow_t:
        class inherits( wForm_t );
              .
              .
              .
          << mainAppWindow_t class declarations >>
              .
              .
              .
        endclass;
```

For each button we place on the form, we've got to have a corresponding data field declaration. Combined with the two data fields appearing in the *004_button3.hla* source file, we have the following declarations at the beginning of the class:

```
    mainAppWindow_t:
```

```
class inherits( wForm_t );

    // We have to add VAR declarations for all our widgets
    // here.

    var
        button1     :wPushButton_p;
        button2     :wPushButton_p;
        button3     :wPushButton_p;
        button4     :wPushButton_p;
        button5     :wPushButton_p;
        button6     :wPushButton_p;
        quitButton  :wPushButton_p;
        showState   :boolean;
        b1Enabled   :boolean;
        align(4);
```

The `button1..button6` and `quitButton` declarations correspond to the `wButton` declarations in the *004_button3.hla* source file.

Every class derived from wForm_t needs to override the `onCreate` and `onClose` methods. The `onClose` method is the code that actually tells Windows to terminate the program (it's part of the boilerplate code that appears in all the tutorial files) and the `onCreate` method is where *004x_button3.hla* puts the code to initialize the object's data field variables and set up the event handlers. Here's the class prototypes for these methods (which immediately follows the data field declarations given above):

```
    // We need to override these:

    override method onClose;
    override method onCreate;
```

The last declaration in the mainAppWindow_t class is the constructor for the class. Here's the procedure declaration for the constructor (and the end of the class declaration):

```
    // Every main application window must have a
    // constructor with the following prototype:

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

endclass;
```

After the class declaration, there is some (basically boilerplate) code that declares a pointer type to the class, declares the VMT, an object of the class type, and a pointer to that object:

```
        mainAppWindow_p :pointer to mainAppWindow_t;

// Must have the following declarations in all (manually written) HOWL apps:

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

The `mainAppWindow` variable is the actual class object for the form. Most apps will never actually use this variable (why use `pmainAppWindow` when `mainAppWindow` is available?), nevertheless it doesn't hurt to declare it just in case it's convenient to use for some purpose.

The next bit of code in *004x_button3.hla* is the constructor for the `mainAppWindow_t` class. Let's take a look at this code in a piece-by-piece fashion:

```
// Here is the constructor we must supply for the mainAppWindow class:

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
);
var
    main               :mainAppWindow_p;

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;
```

The code above is mostly typical of any HLA class constructor. If ESI contains NULL upon entry, this means you've called the constructor as a class constructor (e.g., `mainAppWindow_t.create_mainAppWindow`, notice the "_t" at the end of `mainAppWindow_t`) that tells the constructor you want to create a new object whose storage is allocated on the heap. Generally, this will not be the case because you'll normally call the constructor using the call `mainAppWindow.create_mainAppWindow` (notice the lack of a "_t" at the end of the first `mainAppWindow` in this call). When called this way, ESI will contain the address of the `mainAppWindow` variable upon entry into the constructor.

About the only thing novel about this code (novel may be too strong a word as almost every manually-written HOWL application does this) is the fact that the CL register is loaded with true or false depending on whether the object was initialized on the heap (note that the static declaration of `mainAppWindow` variable initializes the `this.wBase_private.onHeap` field to false automatically when the program is loaded into memory). At a later time in this procedure the program will store the value of CL into the `this.wBase_private.onHeap` data field.

Note that this code accesses private data fields of the `wBase_t` class. In applications, such access is normally forbidden. However, keep in mind that we are writing the class constructor for a HOWL class here, and HOWL class procedures and methods can access the private data fields.

Moving on, the next piece of code handles the generic `wForm_t` object initialization by calling the parent class' constructor:

```
// 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
);
```

The call above is where most of the real initialization work for a generic `wForm_t` object actually gets done.

The next step is to initialize the object's VMT pointer with the address of the `mainAppWindow_t` VMT. Again, this is standard HLA constructor initialization stuff:

```
// Initialize the VMT pointer:

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

Next, we initialize the `this.wBase_private.onHeap` variable with the value saved in the CL register and we also preserve the value of the this pointer (held in ESI) because we're about to call several class constructors that will wipe out ESI's value:

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

mov( cl, this.wBase_private.onHeap );

// Preserve "this" because we're about to make an object call
// that will overwrite this' value:

mov( esi, main );
```

In the *004_button3.hla* source file, we didn't have direct access to the constructor for the `mainAppWindow_t` class, so we stuck the application-specific initialization for that class in the `onCreate` method. However, as we're writing the constructor manually in this example, we may as well put that object initialization code directly in the constructor. So here's the code that initializes the `showState` and `b1Enabled` data fields:

```
// Initialize the showState and enableDisableButton data fields:

mov( false, this.showState );
mov( true, this.b1Enabled );
```

And now, the real fun begins. For each of the buttons on the form (that is, for each of the `wPushButton` HDL declarations appearing in the *004_button3.hla* `wForm..endwForm` statement), we need to call a `wPushButton` constructor and perform various housekeeping activities to add that button to our mainAppWindow form. The next sequence of statements in the constructor, for example, creates the main push button (`button1`) on the form:

```
// The primary push button on the form:

wPushButton_t.create_wPushButton
(
    "button1",                  // Button name
    "Press to change",          // Caption for push button
    this.handle,                // Parent window handle
    10,                         // x position
    10,                         // y position
    125,                        // width
    25,                         // height
    &onClickChange1             // initial "on click" event handler
);
```

Note that upon return from the (class) constructor above, ESI no longer points at the `mainAppWindow` form object, instead it contains the address of the new `wPushButton` object created on the heap. As long as we have easy access to the `button1` object (in ESI), we can write some code to register the `onSetFocus` and `onKillFocus` event handlers (this was done in the `onCreate`

method in the *004_button3.hla* source file because we couldn't insert code directly into the `wForm..endwForm` statement):

```
// Set up the onSetFocus and onKillFocus widgetProcs.

(type wPushButton_t [esi]).set_onSetFocus( &onSetFocus1 );
(type wPushButton_t [esi]).set_onKillFocus( &onKillFocus1 );
```

The last thing we have to do with a new object is store away the object's pointer (currently in ESI) into the button1 data field of the main form.

```
mov( esi, mainAppWindow.button1 );  // Save ptr to new button
main.insertWidget( esi );           // Add button to wForm's widget list.
```

We repeat this process for all the remaining buttons. However, except for button6, there aren't any event handlers to be initialized, so we drop the code that registers the widgetProcs for most of these buttons:

```
// The show/hide button on the form:

wPushButton_t.create_wPushButton
(
    "button2",                  // Button name
    "Hide button 1",            // Caption for push button
    this.handle,                // Parent window handle
    175,                            // x position
    10,                         // y position
    125,                        // width
    25,                         // height
    &hideShowButton             // initial "on click" event handler
);
mov( esi, mainAppWindow.button2 );  // Save ptr to new button
main.insertWidget( esi );           // Add button to wForm's widget list.


// The enable/disable button on the form:

wPushButton_t.create_wPushButton
(
    "button3",                  // Button name
    "Disable button 1",         // Caption for push button
    this.handle,                // Parent window handle
    175,                            // x position
    40,                         // y position
    125,                        // width
    25,                         // height
    &enableDisableButton        // initial "on click" event handler
);
mov( esi, mainAppWindow.button3 );  // Save ptr to new button
main.insertWidget( esi );           // Add button to wForm's widget list.


// The move button on the form:
```

```
wPushButton_t.create_wPushButton
(
    "button4",                  // Button name
    "Move button 1",            // Caption for push button
    this.handle,                // Parent window handle
    175,                        // x position
    70,                         // y position
    125,                        // width
    25,                         // height
    &moveButton                 // initial "on click" event handler
);
mov( esi, mainAppWindow.button4 );  // Save ptr to new button
main.insertWidget( esi );           // Add button to wForm's widget list.


// The resize button on the form:

wPushButton_t.create_wPushButton
(
    "button5",                  // Button name
    "Resize button 1",          // Caption for push button
    this.handle,                // Parent window handle
    175,                        // x position
    100,                        // y position
    125,                        // width
    25,                         // height
    &resizeButton               // initial "on click" event handler
);
mov( esi, mainAppWindow.button5 );  // Save ptr to new button
main.insertWidget( esi );           // Add button to wForm's widget list.
```

The double-click button (`button6`) requires the registration of an `onDblClick` event handler. So the code that initializes this button object contains an extra line of code that handles that chore:

```
// The double-click button on the form:

wPushButton_t.create_wPushButton
(
    "button6",                  // Button name
    "DblClick to Click",        // Caption for push button
    this.handle,                // Parent window handle
    175,                        // x position
    130,                        // y position
    125,                        // width
    25,                         // height
    NULL                        // initial "on click" event handler
);

// Set up the onDblClick event handler:

(type wPushButton_t [esi]).set_onDblClick( &onDblClick );

mov( esi, mainAppWindow.button6 );  // Save ptr to new button
main.insertWidget( esi );           // Add button to wForm's widget list.
```

The creation of the quit button completes the initialization of buttons on the form:

```
    // We need to create a quit button and store the pointer to the
    // new button object in the this.button field on the form:

    wPushButton_t.create_wPushButton
    (
        "quitButton",               // Button name
        "Quit",                     // Caption
        this.handle,                // parent window handle
        450,                        // x position
        525,                        // y position
        125,                        // width
        25,                         // height
        &onQuit                     // "on click" event handler
    );
    mov( esi, mainAppWindow.quitButton );   // Save ptr to new button
    main.insertWidget( esi );               // Add button to wForm's widget list.
```

The last thing a `wForm` (or derived from `wForm`) constructor must do is call the `onCreate` method. As it turns out, `onCreate` is often empty in HOWL applications that manually define the form(after all, you could simply insert whatever code you'd put into `onCreate` in place of this call), but just in case someone decides to stick some code in `onCreate` at a later date, it's a good idea to make this call:

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

end create_mainAppWindow;
```

Once you're done with the constructor, the remaining code in the program is identical to that in the *004_button3.hla* source file with one exception: the `onCreate` method is now empty because we've moved all that code into the constructor:

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

Here's the complete source file for *004x_button3.hla*:

```
// button3-

program button3;
#linker( "comdlg32.lib" )
#linker( "comctl32.lib" )

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

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

const
    applicationName := "Button Demo #3x";
    formX           := w.CW_USEDEFAULT; // Let Windows position this guy
    formY           := w.CW_USEDEFAULT;
    formW           := 600;
    formH           := 600;



type

    // Create a new class for our main application window.
    // All application forms must be derived from wForm_t:

    mainAppWindow_t:
        class inherits( wForm_t );

            // We have to add VAR declarations for all our widgets
            // here.

            var
                button1     :wPushButton_p;
                button2     :wPushButton_p;
                button3     :wPushButton_p;
                button4     :wPushButton_p;
                button5     :wPushButton_p;
                button6     :wPushButton_p;
                quitButton  :wPushButton_p;
                showState   :boolean;
                b1Enabled   :boolean;
                align(4);


            // We need to override these (actually, onClose is the
            // only one that is important):

            override method onClose;
            override method onCreate;

            // Every main application window must have a
            // constructor with the following prototype:

            procedure 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;

// Must have the following declarations in all (manually written) HOWL apps:

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



// Forward declarations for the onClick widgetProcs that we're going to
// call when a button is pressed.

proc onSetFocus1            :widgetProc; @forward;
proc onKillFocus1           :widgetProc; @forward;
proc onClickChange1         :widgetProc; @forward;
proc onClickChange2         :widgetProc; @forward;
proc hideShowButton         :widgetProc; @forward;
proc enableDisableButton    :widgetProc; @forward;
proc moveButton             :widgetProc; @forward;
proc resizeButton           :widgetProc; @forward;
proc onDblClick             :widgetProc; @forward;
proc onQuit                 :widgetProc; @forward;


// Here is the constructor we must supply for the mainAppWindow class:

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
);
var
    main               :mainAppWindow_p;

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 );

// Preserve "this" because we're about to make an object call
// that will overwrite this' value:

mov( esi, main );


// Initialize the showState and enableDisableButton data fields:

mov( false, this.showState );
mov( true, this.b1Enabled );

// The primary push button on the form:

wPushButton_t.create_wPushButton
(
    "button1",                  // Button name
    "Press to change",          // Caption for push button
    this.handle,                // Parent window handle
    10,                         // x position
    10,                         // y position
    125,                        // width
    25,                         // height
    &onClickChange1             // initial "on click" event handler
);
```

```
// Set up the onSetFocus and onKillFocus widgetProcs.

(type wPushButton_t [esi]).set_onSetFocus( &onSetFocus1 );
(type wPushButton_t [esi]).set_onKillFocus( &onKillFocus1 );

mov( esi, mainAppWindow.button1 );  // Save ptr to new button
main.insertWidget( esi );           // Add button to wForm's widget list.




// The show/hide button on the form:

wPushButton_t.create_wPushButton
(
    "button2",                  // Button name
    "Hide button 1",            // Caption for push button
    this.handle,                // Parent window handle
    175,                            // x position
    10,                         // y position
    125,                        // width
    25,                         // height
    &hideShowButton             // initial "on click" event handler
);
mov( esi, mainAppWindow.button2 );  // Save ptr to new button
main.insertWidget( esi );           // Add button to wForm's widget list.




// The enable/disable button on the form:

wPushButton_t.create_wPushButton
(
    "button3",                  // Button name
    "Disable button 1",         // Caption for push button
    this.handle,                // Parent window handle
    175,                            // x position
    40,                         // y position
    125,                        // width
    25,                         // height
    &enableDisableButton        // initial "on click" event handler
);
mov( esi, mainAppWindow.button3 );  // Save ptr to new button
main.insertWidget( esi );           // Add button to wForm's widget list.




// The move button on the form:

wPushButton_t.create_wPushButton
(
    "button4",                  // Button name
    "Move button 1",            // Caption for push button
    this.handle,                // Parent window handle
```

```
    175,                        // x position
    70,                         // y position
    125,                        // width
    25,                         // height
    &moveButton                 // initial "on click" event handler
);
mov( esi, mainAppWindow.button4 );  // Save ptr to new button
main.insertWidget( esi );           // Add button to wForm's widget list.




// The resize button on the form:

wPushButton_t.create_wPushButton
(
    "button5",                  // Button name
    "Resize button 1",          // Caption for push button
    this.handle,                // Parent window handle
    175,                        // x position
    100,                        // y position
    125,                        // width
    25,                         // height
    &resizeButton               // initial "on click" event handler
);
mov( esi, mainAppWindow.button5 );  // Save ptr to new button
main.insertWidget( esi );           // Add button to wForm's widget list.




// The double-click button on the form:

wPushButton_t.create_wPushButton
(
    "button6",                  // Button name
    "DblClick to Click",        // Caption for push button
    this.handle,                // Parent window handle
    175,                        // x position
    130,                        // y position
    125,                        // width
    25,                         // height
    NULL                        // initial "on click" event handler
);

// Set up the onDblClick event handler:

(type wPushButton_t [esi]).set_onDblClick( &onDblClick );

mov( esi, mainAppWindow.button6 );  // Save ptr to new button
main.insertWidget( esi );           // Add button to wForm's widget list.



// We need to create a quit button and store the pointer to the
// new button object in the this.button field on the form:
```

```
    wPushButton_t.create_wPushButton
    (
        "quitButton",               // Button name
        "Quit",                     // Caption
        this.handle,                // parent window handle
        450,                        // x position
        525,                        // y position
        125,                        // width
        25,                         // height
        &onQuit                     // "on click" event handler
    );
    mov( esi, mainAppWindow.quitButton );   // Save ptr to new button
    main.insertWidget( esi );               // Add button to wForm's widget list.

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

end create_mainAppWindow;









// The onDblClick widget proc will handle a double click on button6
// and simulate a single click on button 1.

proc onDblClick:widgetProc;
begin onDblClick;

    mov( mainAppWindow.button1, esi );
    (type wPushButton_t [esi]).click();

end onDblClick;



// The resizeButton widget proc will resize button1 between widths 125 and 150.

proc resizeButton:widgetProc;
begin resizeButton;

    mov( mainAppWindow.button1, esi );
    (type wPushButton_t [esi]).get_width();
    if( eax = 125 ) then

        stdout.put( "Resizing button to width 150" nl );
        (type wPushButton_t [esi]).set_width( 150 );

    else
```

```
        stdout.put( "Resizing button to width 125" nl );
        (type wPushButton_t [esi]).set_width( 125 );

    endif;

end resizeButton;



// The moveButton widget proc will move button1 between y positions 10 and 40.

proc moveButton:widgetProc;
begin moveButton;

    mov( mainAppWindow.button1, esi );
    (type wPushButton_t [esi]).get_y();
    if( eax = 10 ) then

        stdout.put( "Moving button to y-position 40" nl );
        (type wPushButton_t [esi]).set_y( 40 );

    else

        stdout.put( "Moving button to y-position 10" nl );
        (type wPushButton_t [esi]).set_y( 10 );

    endif;

end moveButton;



// The hideShowButton widget proc will hide and show button1.

proc enableDisableButton:widgetProc;
begin enableDisableButton;

    mov( thisPtr, esi );
    if( mainAppWindow.b1Enabled ) then

        (type wPushButton_t [esi]).set_text( "Enable button 1" );
        mov( false, mainAppWindow.b1Enabled );
        stdout.put( "Disabling button 1" nl );
        mov( mainAppWindow.button1, esi );
        (type wPushButton_t [esi]).disable();

    else

        (type wPushButton_t [esi]).set_text( "Disable button 1" );
        mov( true, mainAppWindow.b1Enabled );
        stdout.put( "Enabling button 1" nl );
        mov( mainAppWindow.button1, esi );
        (type wPushButton_t [esi]).enable();

    endif;

end enableDisableButton;
```

```
// The hideShowButton widget proc will hide and show button1.

proc hideShowButton:widgetProc;
begin hideShowButton;

    mov( thisPtr, esi );
    if( mainAppWindow.showState ) then

        (type wPushButton_t [esi]).set_text( "Hide button 1" );
        mov( false, mainAppWindow.showState );
        stdout.put( "Showing button 1" nl );
        mov( mainAppWindow.button1, esi );
        (type wPushButton_t [esi]).show();

    else

        (type wPushButton_t [esi]).set_text( "Show button 1" );
        mov( true, mainAppWindow.showState );
        stdout.put( "Hiding button 1" nl );
        mov( mainAppWindow.button1, esi );
        (type wPushButton_t [esi]).hide();

    endif;

end hideShowButton;



// The onSetFocus and onKillFocus widgetProcs simply print to the console
// what has happened.

proc onSetFocus1:widgetProc;
begin onSetFocus1;

    stdout.put( "Set focus to button 1" nl );

end onSetFocus1;


proc onKillFocus1:widgetProc;
begin onKillFocus1;

    stdout.put( "Shifted focus from button 1" nl );

end onKillFocus1;


// Here's 1 of 2 onClick handlers for button1. This widgetProc
// changes the caption to "Restore caption" and sets the
// onClick pointer to point at the second onClick handler.


proc onClickChange1:widgetProc;
var
    curCaption  :string;
    curCapBuf   :char[256];

begin onClickChange1;
```

```
    str.init( curCapBuf, @size( curCapBuf ));
    mov( eax, curCaption );

    mov( thisPtr, esi );        // ESI already contains this, but just in case...

    // Print the current caption to the console window:

    (type wPushButton_t [esi]).get_text( curCaption );
    stdout.put( "Current caption1: ", curCaption, nl );


    // Change the caption:

    (type wPushButton_t [esi]).set_text( "Restore Button #1" );

    // Point the onClick handler at onClickChange2:

    (type wPushButton_t [esi]).set_onClick( &onClickChange2 );


    // Print the new caption to the console window:

    (type wPushButton_t [esi]).a_get_text();
    stdout.put( "New caption1: ", (type string eax), nl nl );
    str.free( eax );

end onClickChange1;



// Here's 2 of 2 onClick handlers for button1. This widgetProc
// changes the caption back to "Restore caption" and sets the
// onClick pointer to point at the first onClick handler.


proc onClickChange2:widgetProc;
var
    curCaption  :string;
    curCapBuf   :char[256];

begin onClickChange2;

    str.init( curCapBuf, @size( curCapBuf ));
    mov( eax, curCaption );

    mov( thisPtr, esi );        // ESI already contains this, but just in case...

    // Print the current caption to the console window:

    (type wPushButton_t [esi]).get_text( curCaption );
    stdout.put( "Current caption2: ", curCaption, nl );


    // Change the caption:

    (type wPushButton_t [esi]).set_text( "Button #1" );

    // Point the onClick handler at onClickChange1:
```

```
    (type wPushButton_t [esi]).set_onClick( &onClickChange1 );


    // Print the new caption to the console window:

    (type wPushButton_t [esi]).a_get_text();
    stdout.put( "New caption2: ", (type string eax), nl nl );
    str.free( eax );

end onClickChange2;



// Here's the onClick event handler for our quit button on the form.
// This handler will simply quit the application:

proc onQuit:widgetProc;
begin onQuit;

    // Quit the app:

    w.PostQuitMessage( 0 );

end onQuit;




// We'll use the main application form's onCreate method to initialize
// the various buttons on the form.
//
// This could be done in appStart, but better to leave appStart mainly
// as boilerplate code. Also, putting this code here allows us to use
// "this" to access the mainAppWindow fields (a minor convenience).

method mainAppWindow_t.onCreate;
begin onCreate;
end onCreate;




////////////////////////////////////////////////////////////////////////////
//
//
// 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.  Notice the
// override declaration for onClose in the wForm declaration given earlier.
// Without that, mainAppWindow_t would default to using the wVisual_t.onClose
// method (which does nothing).
```

```
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,                  // Form x-coordinate
        formY,                  // Form y-coordinate
        formW,                  // Width
        formH,                  // Height
        howl.bkgColor_g,        // Background color
        true                    // Make visible on creation
    );
    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
// call the HowlMainApp procedure.

begin button3;

   // 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 );

   // Start the HOWL Framework Main Program:

   HowlMainApp();

   // Delete the brush we created earlier:

   w.DeleteObject( howl.bkgBrush_g );

end button3;
```