38 HOWL: The HLA Object Windows Library

The HLA Object Windows Library (HOWL) is an application framework for Microsoft Windows that greatly simplifies GUI application development for Windows in assembly language. It lets you declare forms and widgets that describe the visual layout of a GUI application and then you need only write small procedures that handle events associated with your GUI elements. This is far less work than writing a standard Win32 application and processing all the messages that Windows sends to such an application.

Once you master the basics of HOWL, you can write the GUI portion of a Win32 application in a fraction of the time it would take using standard Win32 API programming techniques. Your programs will be easier to read and easier to maintain, as well.

As its name suggests, the HLA *Object* Windows Library makes extensive use of HLA's object-oriented programming facilities. If you are unfamiliar with object-oriented programming, or are uncomfortable with HLA's implementation of object-oriented programming, you should take a look at the chapter on "Classes and Objects" in *The Art of Assembly Language* and the chapter on Object-Oriented Programming in the HLA reference manual.

Note: Successful use of HOWL requires HLA v2.8 or later. If you have an earlier version, you will need to upgrade to the latest version in order to use HOWL. It should be obvious by now, but HOWL for Windows only supports Microsoft Windows (2000, XP, Vista, and Windows 7; no guarantees on earlier versions).

38.1 The HOWL Application Framework

Most HLA programs (at least to date) are written with a "main program" that controls the activities of that application by calling various procedures and other code within the application; the main program is the "traffic cop" that determines what happens and the sequence of those operations. An application framework (like HOWL), on the other hand, contains its own "main program" that controls the sequence of operations within the program. As an application programmer, you will supply various procedures that the framework will call and your procedures will handle various tasks as requested by the application framework's main program. This paradigm takes a small amount of effort to get used to if you've never written event-driven applications before, but it's fairly easy to master and you should be able to master it in a few short hours.

For technical reasons, your HOWL applications will still be written as an HLA `program`. However, the main program for your HOWL application is trivial; if your program is named "howlDemo", then this is what your main program will look like:

```
begin howlDemo;

  HowlMainApp();

end howlDemo;
```

`HowlMainApp` is the HOWL application framework main program that you must call from your main program to get the application running.[1]

Beyond the main program in your HOWL application, there are two methods and three procedures you must also supply:

- An "onClose" method (usually containing a single statement)
- An "onCreate" method (usually empty)
- An "AppStart" procedure
- An "AppTerminate" procedure
- An "AppException" procedure

Your application will normally contain many more procedures and other code, of course, but every HOWL application will have at least these five procedures and methods.

The first step in creating a HOWL application is to create a subclass of the `wForm_t` type. You could create such a new class using a declaration like the following:

1. In theory, the HOWL framework could require that you create a unit and the HOWL main program could have been linked in automatically. However, you cannot insert any main programs into the HLA Standard Library (where HOWL resides) without creating linkage problems for all other applications. The simple alternative is to require HOWL apps to call the `HowlMainApp` procedure from their main program.

```
myNewForm_t:
  class inherits( wForm_t );
     << any new fields you want to add >>
  endclass;
```

However, in an typical HOWL application you won't create a new `wForm_t` type this way. Instead, you'll use the `wForm..endwForm` declaration to achieve this:

```
wForm( myForm )
  << any new fields you want to add   >>
  << any widget declarations you want >>
endwForm
```

The `wForm..endwForm` statement does several things for you:

- It defines a new class type that is a subclass of `wForm_t`. This new class type is given the name "myForm_t" (substituting whatever name you supply as the argument to `wForm` for "myForm" in this example).
- It creates a static global variable named "p*formname*" (substituting the name you supply in the `wForm` invocation for *formname*) that is a pointer to an object of type *`formname`*`_t` (again, substituting the name you supply in the invocation to `wForm` for *formname*).
- It creates a static global variable named "*formname*" (usual substitution) that is an instance of the class object *`formname`*`_t`.
- It initializes "p*formname*" with the address of the global "*formname*" variable.
- It creates a macro, named *`formname`*`_implementation`, that you can invoke to create a constructor for this new class that initializes all the widgets you declare in the `wForm..endwForm` statement.

You could manually do all of these things that the `wForm..endwForm` declaration does for you, but it's a lot easier and less error prone to to use the `wForm..endwForm` statement. So this documentation will only consider that approach. If you're interested in learning how to manually create new `wForm_t` subclass types, take a look at the macro implementation of `wForm..endwForm` in the *howl.hhf* header file.

In order to provide concrete examples, this documentation will assume that you're supplying the name "myForm" as an argument to the `wForm..endwForm` macro invocation. Please keep in mind that you can use any name you choose (and "myForm" is not a particularly descriptive or good name) in these examples. In general, you might want to consider using your application's name (if you're not already using that as an identifier elsewhere in your program) as the main form name.

With the basic description wForm out of the way, we can now take a look at a complete HOWL application (the generic equivalent of a "hello world" application in the GUI world). The following code appears in pieces in order to explain the purpoes of each piece.

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

The HOWL library makes use of the Windows common controls and common dialogs dynamically linked libraries. The two #linker statements above instruct HLA to emit instructions to the linker to link in these libraries (so you don't have to specify these library names on the HLA compiler command line).

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

By default, HLA emits code to generate displays for all procedures and it also emits code to align the stack at the beginning of each procedure. These options are rarely needed in HOWL programming, so it's a good idea to include the two statements above in order to turn off the code generation for these two features.

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

Like most HLA applications, we'll include the HLA standard library generic header file (*stdlib.hhf*) so we can take advantage of most of the standard library's features. Because stdlib.hhf doesn't automatically include the *howl.hhf* header file, the code needs to explicitly include this header file as well in order to use HOWL features.

```
static
    align( 4 );
    bkgBrush_g  :dword;
    bkgColor_g  :dword;
```

The bkgBrush_g and bkgColor_g variables will hold the background color and brush for our forms. We need to declare these variables before our form declaration because many widgets (controls) will need a background color and the widgets will look better if their background color is the same as the form's.

```
wForm( myForm )
endwForm
```

Here's our wForm declaration. It is important for you to understand that the wForm..endwForm declaration is equivalent to a class declaration in the type section of an HLA program. The wForm macro, in fact, expands in-place) to a type section with a class declaration for myForm_t. There are two things you can place between the wForm and endwForm clauses: class field declarations and invocations of certain context-free macros defined by the wForm macro invocation. We'll take a look at both of these options a little later, for the current example our wForm..endwForm macro invocation is empty because we're just creating an empty formfor our application. Note, and this is very important, that the wForm..endwForm statement must appear at some point in your program where it is legal to begin a type section that defines a class (because this is exactly what wForm..endwForm is going to do).

```
// Implement the mainAppWindow create procedure and object instances:

myForm_implementation();
```

One of the tasks that the wForm..endwForm macro invocation accomplishes is the creation of a macro (named myForm_implementation in this example) that will expand to some code that the wForm..endwForm macro generates. Somewhere in your program you must invoke this macro (myForm_implementation) in order to actually emit the code that the new class you've created requires. If you're wondering why the wForm..endwForm macro doesn't emit this code directly, just keep in mind that the wForm..endwForm declaration often appears in a header file (rather than directly in your main program as it appears here) and if wForm..endwForm automatically emitted this code, it would be emitted in every file that included the header file. This would result in duplicate code (and duplicate external labels). Therefore, the wForm..endwForm statement generates this macro that you must invoke exactly once in your program in order to compile the code that wForm..endwForm generates.

```
method myForm_t.onCreate;
begin onCreate;
end onCreate;

method myForm_t.onClose;
begin onClose;

    w.PostQuitMessage( 0 );

end onClose;
```

The myForm_t class that wForm..endwForm creates will define two methods but it will not generate the code for these methods, you will have to provide the code for these methods. Most of the time, these methods will appear exactly as the two above. The HOWL main program invokes the onCreate method at the very end of the execution of the constructor procedure for the myForm_t class. You could put code in this method to execute immediately after the creation of the new form but, as you'll soon see, you'll most often put this "on creation" code in the appStart procedure. So most of the time the onCreate method will be empty. Why does HOWL generate a call to a method you'll almost always leave empty? Well, this is an artifact of object-oriented programming. The onCreate method is quite useful for other classes that are derived from myForm_t's parent class, so rather than make a special case out of myForm_t (or the other classes), HOWL requires you to write this (usually empty) method and it will call it. Fortunately, this only takes a few bytes of memory and it only gets called once, so there really isn't an efficiency loss for doing this.

When the user of your application clicks on the form's close button, or otherwise tells the application to terminate, HOWL will direct control to the `myForm_t.onClose` method. You can do other things to clean up your application when the program is about to quit, but the main thing you need to do is to tell Windows that the application is quitting and this is done by executing `w.PostQuitMessage( 0 );` As it turns out, you generally won't put any application cleanup code directly in the `onClose` method; there is an `appTerminate` procedure that HOWL will call when your program terminates execution and you'll put all your cleanup code in that procedure.

We've talked about the `appStart` and `appTerminate` procedures, let's take a look at them. A typical `appStart` procedure takes the following form:

```
procedure appStart;
begin appStart;

    push( esi );

    // Create the main application window:

    w.GetSysColor( w.COLOR_MENU );
    mov( eax, bkgColor_g );
    w.CreateSolidBrush( eax );
    mov( eax, bkgBrush_g );
```

The above statements initialize the `bkgColor_g` and `bkgBrush_g` global variables with the color (and corresponding solid brush) of the menu area on a typical Windows' window. Though it's not strictly necessary to have these global variables, they are useful when creating widgets in more complex applications, so most `appStart` procedures will initialize these variables. Note that you must initialize these variables before calling the constructor for the `myForm_t` class because the widget declarations appearing in the `wForm..endwForm` statement will be initialized in the call to the constructor for `myForm`. Note that you don't have to specify the `w.COLOR_MENU` background color. You can specify any RGB color you like. You can use the RGB macro (e.g., RGB(255, 128, 0)) to specify the individual red, green, and blue (respectively) components of the RGB color.

The `wForm( mForm ) .. endwForm` statement appearing earlier in the source file (and the `myForm_implementation` macro invocation) automatically generated a constructor for the `myForm_t` class named `create_myForm`. That declaration also created an instance variable (an object) of type `myForm_t` named `myForm`. In order to initialize and display the form, you must call this constructor with the following (or reasonably compatible) arguments:

```
    myForm.create_myForm
    (
        "My Form",              // Window title
        0,                      // Extended style
        0,                      // Style
        NULL,                   // No parent window
        w.CW_USEDEFAULT,        // x-position on screen
        w.CW_USEDEFAULT,        // y-position on screen
        600,                    // Width
        600,                    // Height
        bkgColor_g,             // Background color
        true                    // Make visible on creation
    );
    mov( esi, pmyForm );        // Save pointer to main window object.
```

The first argument is a string parameter that HOWL will display in the title bar of the application's main form. You can put any string you like here, although you should generally put the program's name in the title bar.

The next two parameters let you extend the window style and extended style. In general, this two parameters will contain zero. HOWL always uses the style (`w.WS_CLIPCHILDREN | w.WS_OVERLAPPEDWINDOW`) and the extended style `w.WS_EX_CONTROLPARENT` when creating windows for a HOWL form. The styles you specify in the second and third constructor arguments will be logically ORed into HOWL's existing styles.

The fourth parameter is the handle of the form's parent form. For main application windows, this argument is always NULL.

The fifth and sixth arguments in the constructor invocation are the x- and y-coordinates on the screen where Windows will position the upper-left-hand corner of the form. You may specify any reasonable values here (within the range of the size of your screen, if you want the form to be visible) or you can specify `w.CW_USEDEFAULT,` in which case Windows will position the window automatically for you. For the position on the screen, the default coordinates are probably good values to use.

The seventh and eighth arguments are the width and height of the window you're creating. You'll have to pick these numbers based on the layout of the widgets on your form. You can also specify `w.CW_USEDEFAULT` for the width and height and Windows will pick values it feels are appropriate. Generally, though, you'll want to explicitly specify the width and height for a typical form.

The ninth parameter is the background color to use for the main client area of the form. This is an RGB color value. As this example demonstrates, it's a good idea to u se the bkgColor_g value in this parameter argument so you can specify that same value (or `bkgBrush_g`) for other elements on the display that have a background color.

```
    // Return main window handle in EAX if you want the system
    // to support control selection via the TAB key. Return
    // NULL in EAX if you don't want to support TAB key selection
    // of the controls:

    mov( myForm.handle, eax );
    pop( esi );

end appStart;
```

The caller to `appStart` checks the return value in EAX to determine whether the form supports tab control selection. If you return NULL in EAX, then the form ignores the tab key during execution. If you return the form's handle (`myForm.handle`), then Windows will support control/widget tab selection on the form.

```
// 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.

    myForm.destroy();
    w.DeleteObject( bkgBrush_g );

end appTerminate;
```

The HOWL system will call the `appTerminate` procedure after the `myForm.onClose` method terminates. The `appTerminate` procedure is a good place to call the class destructor (`myForm.destroy`) and clean up any other resources in use. In the code above, for example, the `appTerminate` procedure deletes the brush resource created in the `appStart` procedure. Note that you should only call the main form destructor in the `appTerminate` procedure, not in the `myForm.onClose` method. The destructor will free up any allocated storage and other resources in use and you shouldn't do this while the object is still active (e.g., while the `onClose` method is executing).

```
// 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;
```

HOWL will call the `appException` procedure if an unhandled exception occurs during the execution of your program. As a general rule, it's a good idea to clean up resouces as best you can before bailing out of the program.

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

begin howlDemo;

    HowlMainApp();

end howlDemo;
```

The main application of a HOWL program, as noted earlier, should simply call the `HowlMainApp` procedure which is the real "main program" of the system.

Here's the complete "trivial HOWL application" withou the annotations:

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

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

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

static
    align( 4 );
    bkgBrush_g  :dword;
    bkgColor_g  :dword;


wForm( myForm )
endwForm

// Implement the mainAppWindow create procedure and object instances:

myForm_implementation();




method myForm_t.onClose;
begin onClose;

    w.PostQuitMessage( 0 );
```

```
end onClose;


method myForm_t.onCreate;
begin onCreate;
end onCreate;



procedure appStart;
begin appStart;

    push( esi );

    // Create the main application window:

    w.GetSysColor( w.COLOR_MENU );
    mov( eax, bkgColor_g );
    w.CreateSolidBrush( eax );
    mov( eax, bkgBrush_g );
    myForm.create_myForm
    (
        "My Form",              // Window title
        0,                      // Extended style
        0,                      // Style
       NULL,                    // No parent window
        w.CW_USEDEFAULT,        // Let Windows position this guy
        w.CW_USEDEFAULT,        //  "    "          "      "    "
        600,                    // Width
        600,                    // Height
        bkgColor_g,             // Background color
        true                    // Make visible on creation
    );
    mov( esi, pmyForm );        // Save pointer to main window object.

    // Return main window handle in EAX if you want the system
    // to support control selection via the TAB key. Return
    // NULL in EAX if you don't want to support TAB key selection
    // of the controls:

    mov( myForm.handle, eax );
    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.

    myForm.destroy();
    w.DeleteObject( bkgBrush_g );

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 howlDemo;

    HowlMainApp();

end howlDemo;
```

Assuming you've named this file "myForm.hla", you can compile and run this program as follows:

```
hla myForm
```

You can run the application from a command-line by typing "myForm" or you can double-click on the icon to run it in a traditional GUI style. Note that if you double-click on the icon, Windows will open up both the GUI window (your form) and a console window. As it turns out, having the console window is useful for debugging purposes, but if you want to ship a release version of your application, you'll probably want to ditch the console window. You can achieve this by using the following command line to compile your application:

```
hla -w myForm
```

When you run the program (either from the command line or by double clicking on an icon), an empty gray window will appear. You can quit the program by clicking on the close box on the title bar.

To understand how the HOWL application framework functions, we need an example that is slightly more complex than the trivial example presented thus far. Consider the following modifications to the previous example:

```
procedure onPress(thisPtr:dword; wParam:dword; lParam:dword ); forward;

wForm( myForm )

        wPushButton
        (
            myButton,           // Field name in mainWindow object
            "Close",            // Caption for push button
            10,                 // x position
            10,                 // y position
```

```
            125,                // width
            25,                 // height
            onPress             // "on click" event handler
        );

endwForm
    .
    .
    .
procedure onPress(thisPtr:dword; wParam:dword; lParam:dword );
begin onPress;

    w.PostQuitMessage( 0 );

end onPress;
```

The `onPress` procedure is an example of an *event handler*. The HOWL code will automatically call various event handlers in response to some system event, such as someone pressing a button on a form. In this example, we're going to place a push button on the formand the HOWL code will call the `onPress` event handler in response to someone pressing that button. How does HOWL know to call `onPress` rather than some other function? Easy, we're going to tell it about `onPress`. The first step is to create a forward declaration for `onPress` that must appear before the `wForm..endwForm` statement. We're going to reference `onPress` inside the `wForm..endwForm` statement, so we have to make sure it's declared before `wForm( myForm )`; the forward declaration achieves this.[2]

The `wPushButton` delcaration appearing inside the `wForm..endwForm` statement tells HOWL to create a push button object on the form. The first argument (`myButton`) is the name of the field that HOWL will create inside the `myForm` class to represent the push button object. This is roughly equivalent to the following declaration in HLA:

```
type
  myForm: class inherits( wForm_t);
     var
         myButton: wPushButton_p;
             .
             .
             .
  endclass;
```

(note that type `wPushButton_p` is defined to be a pointer to `wPushButton_t`.)

The remaining arguments in the wPushButton statement define the appearance of the button on the actual form. The second argument is the string label that HOWL will display on the button. The third through sixth arguments, as the comments state, specfy the x- and y-coordinates and the width and height of the button on the form. Note that the coordinates are relative to the upper-left-hand corner of the *client area* on the form.[3] The last argument is the one of most interest to us here. This is the name of the "on click" event handler that HOWL will call when you press the push button on the form.

All widget event handlers are "widget procs". This means that they are procedures with the following parameter list:

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

Note that `widgetProc` procedures are not class procedures or methods, they are standard procedures. Specifically, this means that you can't use the `this` or `super` reserved words inside a `widgetProc`. However, note that a `widgetProc` does have a `thisPtr` parameter. This parameter will contain the address of the object that signaled the event (that is, `thisPtr` will point at the `myButton` object in this example when the user presses the button and HOWL calls the `onPress` procedure). In this example, we'll not use `thisPtr`, but in more

2. You could also put the entire `onPress` procedure before the `wForm` declaration, but generally it's better programming style to put all procedure code after the declarations (like `wForm`), hence the use of the forward declaration in this code.

3. The client area does not include the title bar nor the menus. If the form is tabbed, it doesn't include the area holding the tabs, either.

complex examples having a pointer to the object turns out to be important. The `wParam` and `lParam` arguments are passed from Windows message procedure on to the `widgetProc` unchanged. For certain widget types, these arguments contain important data. In the current example, we ignore these arguments as well.

For the `onPress` procedure in this example, we simply want to quit the program when the user presses the "close" button. Therefore, this `onPress` procedure executes the same code as the main form's `onClose` method, that is: `w.PostQuitMessage( 0 );` If you compile this code and run it, you can quit the program by pressing on the "close" button.

In most cases, these event handler procedures are how HOWL communicates with your code. Whenever the user initiates some event by interacting with the widgets on the form, HOWL calls the corresponding widget procedure(s) to let your program handle the event.

38.2 The HOWL Declarative Language

In the previous section you saw a couple of simple examples of the HOWL declarative language. The HOWL declarative language is the set of statements that are legal inside a `wForm..endwForm` sequence. The wPushButton statement was a good example of a HOWL declarative language statement. There are many other widget-defining statements present in the HOWL declarative language. This section will describe those statements.

Before describing the actual statements that define widgets, you should note that the `wForm..endwForm` statement is actually a class declaration in the HLA language. So in addition to all the legal declarative statements, you can also put any legal class declarations inside a `wForm..endwForm` statement. For example, if you want to communicate some data between various widgets on a form, one way to achieve this is by placing some class `var` declarations inside the `wForm..endwForm` statement:

```
wForm( myForm )
  var
      someData  :dword;
          .
          .
          .
endwForm
```

Your event handlers and other code and refer to this data field using the syntax `myForm.someData`.

Of course, you can add any other legal class objects into this declaration including class constants, procedures, methods, and iterators.

The *howl.hhf* header file implements the HOWL declarative language using an HLA context-free macro. The `wForm..endwForm` macro declaration includes various `#keyword` macros (like `wPushButton`) that expand into appropriate declarations in the class (and via some macro magic, store away code to create the constructor procedure for the class). When you look at some HOWL delcarative code, it's easy to forget that you're looking at a *declaration*, not at *executable code*. It's easy to think that you should be able to do something like the following:

```
wForm( myForm )

        mov( btnXPosn, eax );
        add( 10, eax );
        mov( btnYPosn, ecx );
        add( 10, ecx );
        wPushButton
        (
            myButton,           // Field name in mainWindow object
            "Close",            // Caption for push button
            eax,                // x position
            ecx,                // y position
            125,                // width
            25,                 // height
            onPress             // "on click" event handler
        );

endwForm
```

However, this won't work. Don't forget that the statements inside the wForm..endwForm declaration are emitted inside a class declaration. Stray statements such as "mov(btnXPosn, eax);" cannot appear inside a class declaration.

It is possible to sneak certain statements into the your widget declarations. Except for the first parameter in most widget declarations (which is the widget object's name in the class declaration), HOWL records the parameter's value and "plays it back" when generating the code for the form's class constructor. This means you can sneak in some code if that code is appropriate for a parameter in a procedure call. Consider the following:

```
wForm( myForm )

        wPushButton
```

```
        (
            myButton,           // Field name in mainWindow object
            "Close",            // Caption for push button
            returns
            ({
                mov( btnXPosn, eax );
                add( 10, eax );
            }, "eax" ),
            10,                 // y position
            125,                // width
            25,                 // height
            onPress             // "on click" event handler
        );

endwForm
```

You may specify global variables as arguments to a widget declaration, but you have to ensure that you've declared the global object prior to the `wForm` statement at that you've initialized the global object before calling the constructor for that form.

The following subsections describe the `wForm..endwForm` statement as well as all the widgets that may appear within a `wForm..endwForm` declaration.

38.2.1 wForm

```
wForm( <<formname>> )
  << widget and class field declarations >>
endwForm
```

The `wForm..endwForm` statement declares a form (or window) for an application. Every application will have at least one of these statements. The single argument is the name associated with the form. HOWL takes this name and generates five distinct program entities from it:

- A class type named *`formname`*`_t` that represents the form's type and holds the declarations for all the widgets on the form.
- A data type named *formname*_p,
- A global variable named *`formname`*, of type *`formname`*`_t`, that is an instance of the form object.
- A global variable named `p`*`formname`* of type *`formname`*`_p` that is initialized with the address of the *`formname`* object.
- A constructor (class procedure) of the name *`formname`*`_t.create_`*`formname`*. You will call this constructor to initialize the *`formname`* variable object.

Most programs will use the *`formname`* variable directly and ignore the `p`*`formname`* variable. However, `p`*`formname`* is available if having a pointer to the object is more convenient that the object itself.

The constructor for the object has the following prototype:

```
procedure formname_t.create_formname
(
    caption     :string;
    exStyle     :dword;
    style       :dword;
    parent      :dword;
    x           :dword;
    y           :dword;
    width       :dword;
    height      :dword;
    fillColor   :dword;
    visible     :boolean
);
```

caption HOWL displays this string in the title bar of the form's window.

exStyle HOWL logically ORs this MS Windows "extended window style" (w.WS_EX_* constants) with w.WS_EX_CONTROLPARENT when creating the window for the form.

style HOWL logically ORs this MS Windows "window style" (w.WS_* constants) with w.WS_CLIPCHILDREN | w.WS_OVERLAPPEDWINDOW when creating the window for the form.

parent This is the handle of the parent window for this form. As most `wForm` objects do not have parent windows, this parameter should contain NULL.

x The x-coordinate on the screen of the upper-left-hand corner of the form.

y The y-coordinate on the screen of the upper-left-hand corner of the form.

width The width of the form (in pixels)

height The height of the form (in pixels)

fillColor This is an RGB value that specifies the background color for the form. The red component is the L.O. eight bits of this value, the green component is in bits 8..15 of this value, and the blue component is in bits 16..23. Bits 24..31 should contain zero. The howl.hhf header file defines a macro, RGB, that lets you assemble an RGB value from three constants: RGB(redConst, greenConst, blueConst). For example, RGB(255, 0, 0) corresponds to red will a full intensity.

visible This is a boolean constant that specifies whether the form will be created in a visible (true) or hidden (false) state. You can always call the *`formname`*`.show()` method to make a form visible or the *`formname`*`.hide()` method to hide the form at run-time. For the main application window, this field normally contains true.

Generally, *`formname`* objects are singletons. That is, you typically create only a single instance of a *`formname`* object. However, as *`formname_t`* is a standard HLA data type, there is nothing stopping you from creating multiple objects of this type. Most of the type, however, it would be somewhat confusing to have two instances of the same form on the display at one time. In some instances this is reasonable. For example, if you have a text editor form and you want to allow the user to edit multiple files (or multiple views of the same file) concurrently, it might make sense to have two instances of the same window on the screen at one time.

`wForm..endwForm` declarations are not recursive. That is, you cannot embed a `wForm` inside another `wForm` declaration. `wForm` windows are main application forms and support menus, tabs, and other facilities only possible on the main application form, hence the restriction.

`wForm` objects are examples of *containers* in HOWL. A container, as its name suggests, may contain other objects (specifically widget objects). There are a couple of different kinds of containers in HOWL, but `wForm` contains are special because they can contain two things that other containers cannot: menus and tabs. Therefore, this document will describe these widgets next

38.2.2 wMainMenu..endwMainMenu

```
wMainMenu

  << main menu widget declarations >>

endwMainMenu
```

A `wForm` object can contain an optional *main menu* widget. This consists of the `wMainMenu..endwMainMenu` statement. If a form contains a main menu widget, the main menu widget must be the first widget declaration appearing on the form. E.g.,

```
wForm( myForm )

    wMainMenu

        << menu item declarations >>

    endwMainMenu

    << other widget declarations >>

endwForm
```

Within the wMainMenu..endwMainMenu declaration you define the items that appear on the menu. These include menu items, submenus, and separators. The following subsections describe each of these items.

A `wMainMenu` object can be thought of as a container object, albeit a very specialized one. `wMainMenu` objects can contain `wMenuItems, wMenuSeparators`, and `wSubMenu` objects. Unlike normal containers, however, `wMainMenu` objects cannot contain arbitrary HOWL objects.

38.2.2.1 wMenuItem

A `wMenuItem` declaration defines a single menu item in a main menu or a submenu. This declaration is only legal within a `wMainMenu..endwMainMenu` or `wSubMenu..endwSubMenu` declaration.

```
wMenuItem
(
    menuItemName,
    menuItemChecked,
    menuString,
    menuHandler
);
```

The `menuItemName` argument is an identifier that HOWL inserts into the wForm object to represent this particular menu item. You generally won't directly refer to this identifier in your programs, but HOWL requires a field name so you must supply a unique (to the class) identifier here.

The `menuItemCheck` argument is a boolean constant that specifies whether the menu item can contain a check mark next to it in the menu display. If this is true, then the menu item will have the ability to display (or not display) a checkmark next to the menu item. If this field is false, then the menu item will not have this ability. See the discussion of the `wMenuItem_t.checked` method to see how to set or clear this checkmark.

The `menuString` argument is a string constant that specifies the text that HOWL will display for the menu item.

The `menuHandler` argument is either NULL or the name of a `widgetProc` that HOWL will call when the user selects this particular menu item. If the argument is NULL, HOWL will ignore the selection of the menu item.

HOWL displays menu items across the menu bar on a `wForm` (the menu bar appears immediately below the title bar in the window). Generally, most main menu items are actually submenus, though straight menu items are also legal in and main menu.

38.2.2.2 wMenuSeparator

```
wMenuSeparator
```

The `wMenuSeparator` declaration should only appear in a submenu (that is, within a `wSubMenu..endwSubMenu` declaration). This draws a horizontal bar across the menu to separate sets of menu items in a submenu.

38.2.2.3 wSubMenu..endwSubMenu

```
wSubenu

    << menu item declarations >>

endwSubmenu
```

Submenu item declarations are syntactically similar to main menu declarations. However, submenus must always appear inside another menu declaration (either a `wMainMenu..endwMainMenu` or some other `wSubMenu..endwSubmenu` declaration). Unlike `wMenuItem` declarations, there is no handler associated with a submenu. HOWL (and Windows) automatically handles all the processing associated with a submenu.

38.2.2.4 Menu Example

Here's a complete menu declaration, including submenus within submenus and menu items in the main menu:

```
wForm( mainAppWindow );

  wMainMenu;

      wSubMenu( menu_1, "menu1" );

          wMenuItem( menu_1_1, false, "Item_1_1", handler_1_1 );
          wMenuItem( menu_1_2, true, "Item_1_2", handler_1_2 );
          wMenuItem( menu_1_3, true, "Item_1_3", handler_1_3 );
          wMenuSeparator;
          wMenuItem( menu_exit, false, "Exit", exitHandler );

      endwSubMenu;

      wSubMenu( menu_2, "menu2" );

          wMenuItem( menu_2_1, false, "Item_2_1", handler_2_1 );
          wSubMenu( menu_2_2, "Menu_2_2" );

              wMenuItem( menu_2_2_1, false, "Item_2_2_1", handler_2_2_1 );
              wMenuItem( menu_2_2_2, false, "Item_2_2_2", handler_2_2_2 );

          endwSubMenu;

      endwSubMenu;

      wMenuItem( menu_3, false, "menu3", handler_3 );

  endwMainMenu;

      << other widget declarations >>

endwForm
```

38.2.3 wTab

```
wTab
(
     tabName,     // identifier
     tabString,   // string
     tabHandler,  // widgetProc name or NULL
     bkgColor     // RGB color
)
```

By default, a `wForm` object provides a single surface to which you can attach widgets. The `wTab` declaration allows you to specify multiple surfaces on a form, each user-selectable by clicking on a tab at the top of the form. Like `wForm` objects, `wTab` objects are containers and may contain all the same widgets (except `wMainMenu` items). If you utilize tabs on a `wForm` object, the first `wTab` declaration must appear after the `wMainMenu` declaration (if any) and before any other widgets, e.g.,

```
wForm( mainAppWindow );

  wMainMenu;

        wMenuItem( exitMenu, false, "exit", exitHandler );

  endwMainMenu;

  wTab( tab1, "tab1", NULL, bkgColor_g );

      << widgets associated with tab1 >>
          .
```

```
            .
            .
```

The declarations for widgets associated with a tab appear immediately after that tab up to the next tab declaration or the `endwForm` clause. Most forms, if they use tabs, will have at least two tabs. Here's an example declaration of a form with two tabs:

```
wForm( mainAppWindow );

   wMainMenu;

          wMenuItem( exitMenu, false, "exit", exitHandler );

   endwMainMenu;

   wTab( tab1, "tab1", NULL, bkgColor_g );

      wPushButton
      (
         buttonOnTab1,    // Identifier for button
         "Tab1 Button",   // Caption for push button
         10,              // x position
         10,              // y position
         125,             // width
         25,              // height
         onClick1         // "on click" event handler
      );


   wTab( tab2, "tab2", NULL, bkgColor_g );

      wPushButton
      (
         buttonOnTab2,    // Identifier for button
         "Tab2 Button",   // Caption for push button
         10,              // x position
         10,              // y position
         125,             // width
         25,              // height
         onClick2         // "on click" event handler
      );

endwForm
```

This example creates two tab pages on the form, each with on button on the respective forms.

HOWL and Windows automatically handle switching from one form to the other when the user clicks on the tabs.

Note that if you place on or more tabs on a form, the size of the client area (where you can put other widgets) is reduced by the size of the tabs bar at the top of the form.

38.2.4 Check Boxes

HOWL supports four types of check boxes: standard check boxes (wCheckBox), three-state check boxes (wCheckBox3), left-text check boxes (wCheckBoxLT), and three-state left-text checkboxes (wCheckBox3LT).

The non-three-state checkboxes alternate between two states when the user clicks on the check box (or its caption): checked and unchecked. The three-state check boxes alternate between three states: unchecked, checked, and grayed (don't care).

The standard (non-LT) checkboxes draw their check boxes immediately to the left of the caption (that is, the text is to the right of the check box). The LT (left text) check boxes draw their text to the left of the check box.

CheckBox declarations let you specify an "onClick" event handler that HOWL will call whenever the user clicks on a checkbox and changes its state. This argument should either be the name of a `widgetProc` procedure

or NULL (if you don't want HOWL to call any procedure, which is actually a common occurrence with checkboxes).

38.2.4.1 wCheckBox

```
wCheckBox
(
  checkBoxID,      // Identifier for checkbox
  caption,         // Caption for checkbox
  x,               // x position on form
  y,               // y position on form
  w,               // width
  h,               // height
  onClick          // "on click" event handler (or NULL)
);
```

This declaration creates a standard checkbox on the form.

38.2.4.2 wCheckBox3

```
wCheckBox3
(
  checkBoxID,      // Identifier for checkbox
  caption,         // Caption for checkbox
  x,               // x position on form
  y,               // y position on form
  w,               // width
  h,               // height
  onClick          // "on click" event handler (or NULL)
);
```

This declaration creates a three-state checkbox on the form.

38.2.4.3 wCheckBox3LT

```
wCheckBox3LT
(
  checkBoxID,      // Identifier for checkbox
  caption,         // Caption for checkbox
  x,               // x position on form
  y,               // y position on form
  w,               // width
  h,               // height
  onClick          // "on click" event handler (or NULL)
);
```

This declaration creates a three-state, left-text, checkbox on the form.

38.2.4.4 wCheckBoxLT

```
wCheckBoxLT
(
  checkBoxID,      // Identifier for checkbox
  caption,         // Caption for checkbox
  x,               // x position on form
  y,               // y position on form
  w,               // width
  h,               // height
  onClick          // "on click" event handler (or NULL)
);
```

This declaration creates a left-text checkbox on the form.

38.2.5 wComboBox

A combobox is a combination editBox, listBox, and pull-down menu. The user can type text directly into the editBox section of the combo box or click on the arrow on the right side of the widget to open up a pull-down menu from which the user can select an item.

```
wComboBox
(
  comboBoxID,      // ComboBox name (an identifier)
  "combo box",     // Initial string in the edit box (usually and empty string)
  x,               // x
  y,               // y
  w,               // width
  h,               // height
  sorted,          // true or false
  onCBSelChange,   // onSelChange handler (or NULL)
  "comboBox1",     // List of initial string values for the list
  "comboBox2",     // This list may contain zero or more items.
  "comboBox3"
);
```

The `comboBoxID` argument is the HLA identifier name that HOWL will use for the comboBox object within the wForm declaration; this name should be unique within the form declaration.

The second argument is a string that HOWL will use as the default value of the edit box field when the form is first created. Most often, this will be the empty string. Note that once the user enters data into the edit box or selects and entry from the pull-down menu list, the initial string value is lost.

The `x, y, w,` and `h` fields specify the position and size of the combo box on the form.

The `sorted` field is a boolean value that determines whether the fields of the pull-down menu list are sorted or remain in their "inserted" order. If this field is true, then Windows will sort each entry you add to the list (including the initial entries). If this field is false, then Windows leaves the entries in the order that you add them. Note that it is certainly possible to add and delete fields while the program is running; see the discussion of the wComboBox_t type later in this document. Most often, you'll probably specify false for this field.

The `onCBSelChange` field lets you specify an "on selection change" event handler or NULL if you don't want HOWL to invoke an event handler when the selection change. Normally, you'll put NULL in this field because you'll normally read the text from the widget when you press some other button or when some other event occurs, not when the user changes the text selection in the edit box or selects some entry from the pull-down menu list.

The remaining entries in the `wComboBox` declaration are optional. These entries, if present, must all be string constants that HOWL will use to populate the pull-down menu list. Note that you can add strings to the list at run time, so you don't have to populate the list at declaration time. However, for many lists you'll know the items the user can select from at design time, so you can fill in those entries in the `wComboBox` declaration.

38.2.6 wDragListBox

A `wDragListBox` object is similar to a `wListBox` (list box) object that provides the user with the ability to rearrange items in the list box. The declaration of a `wDragListBox` is

```
wDragListBox
(
   dlName,          // DragListBox name (HLA identifier)
   x,               // x-coordinate
   y,               // y-coordinate
   w,               // width
   h,               // height
   onListBoxClick,  // onClick handler
   "DragListBox1",  // Initial list population (can be empty)
   "DragListBox2",
   "DragListBox3"
);
```

The dlName argument is the name that HOWL will use in the wForm declaration for this wDragListBox object. This name must be unique within the wForm declaration.

The x, y, w, and h arguments specify the position and size of the wDragListBox object on the form.

The onListBoxClick argument is the name of a widgetProc that HOWL will call whenever the user clicks on one of the list items. This field can be NULL, in which case HOWL won't bother to call any procedure when the user clicks on an item (this is actually a common situation; usually the program will determine the currently selected item in a wDragListBox when some other event occurs, and ignore the immediate changes that might occur in a wDragListBox object).

The remaining objects are optional. If present, they must all be strings and the wDragListBox declaration uses these strings to initially populate the wDragListBox object.

38.2.7 wEditBox

A wEditBox object allows the user to enter string data into a program.

```
wEditBox
(
    ebName,       // HLA identifier for this object
    InitialText,  // Initial text for edit box
    x,            // x position
    y,            // y position
    w,            // width
    h,            // height
    s,            // style
    onChange      // onChange handler (can be NULL)
);
```

The ebName argument is the name that HOWL will use in the wForm declaration for this wEditBox object. This name must be unique within the wForm declaration.

The InitialText argument is a string (usually empty) that HOWL uses to initialize the edit box's text field when the form is first created.

The x, y, w, and h arguments specify the position and size of the wEditBox object on the form.

The s argument is the edit box style. This is any of the following edit box styles that HOWL logically ORs with the w.AUTOHSCROLL style:

w.ES_AUTOHSCROLL — Automatically scrolls text to the right by 10 characters when the user types a character at the end of the line. When the user presses the ENTER key, the control scrolls all text back to position zero.

w.ES_AUTOVSCROLL — Automatically scrolls text up one page when the user presses the ENTER key on the last line.

w.ES_CENTER — Centers text in a multiline edit control.

w.ES_LEFT — Left-aligns text.

w.ES_LOWERCASE — Converts all characters to lowercase as they are typed into the edit control.

w.ES_MULTILINE — Designates a multiline edit control. The default is single-line edit control.

When the multiline edit control is in a dialog box, the default response to pressing the ENTER key is to activate the default button. To use the ENTER key as a carriage return, use the ES_WANTRETURN style.

When the multiline edit control is not in a dialog box and the ES_AUTOVSCROLL style is specified, the edit control shows as many lines as possible and scrolls vertically when the user presses the ENTER key. If you do not specify ES_AUTOVSCROLL, the edit control shows as many lines as possible and beeps if the user presses the ENTER key when no more lines can be displayed.

If you specify the ES_AUTOHSCROLL style, the multiline edit control automatically scrolls horizontally when the caret goes past the right edge of the control. To start a new line, the user must press the ENTER key. If you do not specify ES_AUTOHSCROLL, the control automatically wraps words to the

beginning of the next line when necessary. A new line is also started if the user presses the ENTER key. The window size determines the position of the word wrap. If the window size changes, the word wrapping position changes and the text is redisplayed.

Multiline edit controls can have scroll bars. An edit control with scroll bars processes its own scroll bar messages. Note that edit controls without scroll bars scroll as described in the previous paragraphs and process any scroll messages sent by the parent window.

`w.ES_NOHIDESEL`	Negates the default behavior for an edit control. The default behavior hides the selection when the control loses the input focus and inverts the selection when the control receives the input focus. If you specify ES_NOHIDESEL, the selected text is inverted, even if the control does not have the focus.
`w.ES_NUMBER`	Allows only digits to be entered into the edit control.
`w.ES_OEMCONVERT`	Converts text entered in the edit control. The text is converted from the Windows character set to the OEM character set and then back to the Windows set. This ensures proper character conversion when the application calls the CharToOem function to convert a Windows string in the edit control to OEM characters. This style is most useful for edit controls that contain filenames.
`w.ES_PASSWORD`	Displays an asterisk (*) for each character typed into the edit control. You can use the EM_SETPASSWORDCHAR message to change the character that is displayed.
`w.ES_READONLY`	Prevents the user from typing or editing text in the edit control.
`w.ES_RIGHT`	Right-aligns text in a multiline edit control.
`w.ES_UPPERCASE`	Converts all characters to uppercase as they are typed into the edit control.
`w.ES_WANTRETURN`	Specifies that a carriage return be inserted when the user presses the ENTER key while entering text into a multiline edit control in a dialog box. If you do not specify this style, pressing the ENTER key has the same effect as pressing the dialog box's default push button. This style has no effect on a single-line edit control.

The `onChange` argument is the name of a `widgetProc` that HOWL will call whenever the user changes any text in the edit box. This field can be NULL, in which case HOWL won't bother to call any procedure when the user changes the text (this is actually a common situation; usually the program will determine the currently selected item in a `wEditBox` when some other event occurs, and ignore the immediate changes that might occur in a `wEditBox` object). Note that if onChange is non-NULL, then HOWL will call the `widgetProc` any time there is a single-character change to the edit box; this is probably more often than you'd like, which is why this field is generally NULL and applications simply read the data from the edit box when some other event occurs.

38.2.8 wEllipse

```
wEllipse
(
    ellipseName,     // HLA identifier
    x,               // x
    y,               // y
    w,               // width
    h,               // height
    lineColor,       // linecolor (RGB)
    fillColor,       // Ellipse interior color (RGB)
    bkgColor         // Ellipse exterior color (RGB)
)
```

The `ellipseName` argument is the identifier name that HOWL uses in the `wForm` declaration for the ellipse object. This name must be unique within the `wForm` declaration.

The x, y, w, and h fields specify the coordinates and sizes for the bounding box surrounding the ellipse. If these coordinates and sizes form a square, then you'll draw a circle on the form.

The lineColor argument specifies an RGB value that HOWL uses when drawing the outline of the ellipse (the pen color). wEllipse objects always draw the outline with a solid line.

The fillColor argument specifies an RGB value that HOWL uses to fill the interior of the ellipse. wEllipse objects always fill the interior with a solid brush (color).

The bkgColor argument specifies an RGB value that HOWL uses to fill the retangular area described by x, y, w, and h that is outside the ellipse. As a general rule, this should be the same color as the background color for the form unless you want a visible rectangle surrounding the ellipse.

38.2.9 wIcon

The wIcon declaration places an icon object on the form.

```
wIcon
(
    iconIdentifier,             // icon name (HLA identifier)
    IconResourceStr,            // icon resource value
    x,                          // x
    y,                          // y
    w,                          // width
    h,                          // height
    bkgColor                    // background color
)
```

The iconIdentifier field is an HLA identifier that HOWL uses in the form declaration for this particular icon. This name must be unique within the wForm declaration.

The IconResourceStr argument is either a string containing the name of an internal icon resource value or a constant that is less than $1_0000 (that specifies a system icon). If this is a string, it is not a filename for the icon, rather it is a resource ID produced by a resource compiler. See the HLA examples (HOWL directory) for examples that show how to use the resource compiler to produce icons for a program. If this is a value less than $1_0000, then it must be one of the following values:

- w.IDI_APPLICATION
- w.IDI_ASTERISK
- w.IDI_EXCLAMATION
- w.IDI_HAND
- w.IDI_QUESTION
- w.IDI_WINLOGO

Because of syntactical issues with the HLA macro language, if you want to specify these constants as the IconResourceStr argument (which normally must be a string), the best way to do this is to use instruction composition thusly:

```
wIcon
(
    icon1,                          // icon name
    mov( w.IDI_APPLICATION, eax),   // icon resource value
    10,                             // x
    440,                            // y
    32,                             // width
    32,                             // height
    bkgColor_g                      // background color
)
```

The x, y, w, and h fields specify the coordinates and sizes for the bounding box surrounding the icon. If this bounding box is too small for the icon, portions of the icon will be clipped. If this bounding box is too big for the icon, then HOWL will fill the extra area with the background color.

The bkgColor argument specifies the background color that HOWL will use to fill the bounding box for the icon if the icon itself is smaller than the bounding box specified by the x, y, w, and h fields.

38.2.10 wGroupBox..endwGroupBox

A `wGroupBox` object is a container. It draws a rectangular box on a form that contains other objects. All the widgets you declare between a `wGroupBox` statement and the corresponding `endwGroupBox` terminator will be contained by the group box (and can be treated as a whole) at run time.

Note that placing `wRadioButtons` within a `wGroupBox` object does not automatically create a set of radio set buttons. See the `wRadioSet` declaration for that purpose. `wGroupBox` objects really exist just to make the form look pretty.

```
wGroupBox
(
    groupBoxID,      // HLA identifier
    Caption,         // String caption for group box
    x,               // x position
    y,               // y position
    w,               // width
    h                // height
)

    <<Other widget declarations, not including Radio Sets >>

endwGroupBox
```

38.2.11 wLabel

```
wLabel
(
    labelID,         // HLA identifier
    labelString,     // Label string
    x,               // x
    y,               // y
    w,               // width
    y,               // height
    style,           // Alignment and style
    textColor,       // Foreground color
    bkgColor         // Background color
);
```

The `wLabel` declaration lets you place a text string on the form.

The `labelID` argument is the name of the wLabel field within the form's class. This name must be unique within the `wForm` declaration.

The `x, y, w,` and `h` fields specify the coordinates and sizes for the bounding box surrounding the label's string. If this bounding box is too small for the string, portions of the string will be clipped. If this bounding box is too big for the icon, then HOWL will fill the extra area with the background color.

The style field is one or more of the following Windows constants logically-ORed together:

w.DT_BOTTOM	Bottom-justifies text. This value must be combined with DT_SINGLELINE.
w.DT_CENTER	Centers text horizontally.
w.DT_EXPANDTABS	Expands tab characters. The default number of characters per tab is eight.
w.DT_LEFT	Aligns text to the left.
w.DT_NOPREFIX	Turns off processing of prefix characters. Normally, DrawText interprets the mnemonic-prefix character & as a directive to underscore the character that follows, and the mnemonic-prefix characters && as a directive to print a single &. By specifying DT_NOPREFIX, this processing is turned off.
w.DT_RIGHT	Aligns text to the right.
w.DT_SINGLELINE	Displays text on a single line only. Carriage returns and linefeeds do not break the line.

w.DT_TOP	Top-justifies text (single line only).
w.DT_VCENTER	Centers text vertically (single line only).
w.DT_WORDBREAK	Breaks words. Lines are automatically broken between words if a word would extend past the edge of the rectangle specified by the lpRect parameter. A carriage return-linefeed sequence also breaks the line.

For example, to vertically and horizontally center a string within the wLabel bounding box, you would use the following constant for the style field:

```
w.DT_CENTER | w.DT_VCENTER | w.DT_SINGLELINE
```

The `textColor` and `bkgColor` fields are RGB values that specify the (solid) colors used to draw the text and the backgroun for the text. Usually the text's background color is the same as the form's background color unless you are trying to create a special effect.

38.2.12 wListBox

A `wListBox` object contains a sequence of strings from which the user can select a single entry. The declaration of a `wListBox` is

```
wListBox
(
    lbName,          // ListBox name (HLA identifier)
    x,               // x-coordinate
    y,               // y-coordinate
    w,               // width
    h,               // height
    sort,            // true or false
    onListBoxClick,  // onClick handler
    "ListBox1",      // Initial list population (can be empty)
    "ListBox2",
    "ListBox3"
);
```

The `lbName` argument is the name that HOWL will use in the `wForm` declaration for this `wListBox` object. This name must be unique within the `wForm` declaration.

The `x, y, w,` and `h` arguments specify the position and size of the `wListBox` object on the form.

The `onListBoxClick` argument is the name of a `widgetProc` that HOWL will call whenever the user clicks on one of the list items. This field can be NULL, in which case HOWL won't bother to call any procedure when the user clicks on an item (this is actually a common situation; usually the program will determine the currently selected item in a `wListBox` when some other event occurs).

The remaining objects are optional. If present, they must all be strings and the wListBox declaration uses these strings to initially populate the `wListBox` object.

38.2.13 wPasswdBox

A wPasswdBox object is very similar to a wEditBox object insofar as it allows the user to enter a string of text onto the form. The difference is that the wPasswdBox object displays asterisks (or some other character) when the user types a string into the editbox. This shields sensitive information from prying eyes.

```
wPasswdBox
(
    pbName,       // HLA identifier for this object
    IntitialText, // Initial text for edit box
    x,            // x position
    y,            // y position
    w,            // width
    h,            // height
```

```
        s,          // style
        onChange    // onChange handler (can be NULL)
    );
```

The pbName argument is the name that HOWL will use in the wForm declaration for this wPasswdBox object. This name must be unique within the wForm declaration.

The InitialText argument is a string (usually empty) that HOWL uses to initialize the password box's text field when the form is first created. This is almost always the empty string.

The x, y, w, and h arguments specify the position and size of the wPasswdBox object on the form.

The s parameter is the Windows edit box style that HOWL logically ORs with the value (w.ES_AUTOHSCROLL | w.ES_PASSWORD). See the discussion of the available styles in the description of the wEditBox object.

The onChange argument is the name of a widgetProc that HOWL will call whenever the user changes any text in the password box. This field can be NULL, in which case HOWL won't bother to call any procedure when the user changes the text (this is actually a common situation; usually the program will determine the currently selected item in a wPasswdBox when some other event occurs, and ignore the immediate changes that might occur in a wPasswdBox object). Note that if onChange is non-NULL, then HOWL will call the widgetProc any time there is a single-character change to the password box; this is probably more often than you'd like, which is why this field is generally NULL and applications simply read the data from the password box when some other event occurs.

38.2.14 wPie

The wPie declaration defines a graphic object on the form that is a "pie slice", that is, a portion of a pie graph.

```
    wPie
    (
        pieName,        // HLA identifier
        x,              // x
        y,              // y
        w,              // width
        h,              // height
        startAngle,     // Starting handle (in degrees)
        endAngle,       // Ending angle (in degrees)
        lineColor,      // linecolor (RGB)
        fillColor,      // Ellipse interior color (RGB)
        bkgColor        // Ellipse exterior color (RGB)
    )
```

The pieName argument is the identifier name that HOWL uses in the wForm declaration for the pie slice object. This name must be unique within the wForm declaration.

The x, y, w, and h fields specify the coordinates and sizes for the bounding box surrounding the pie slice.

The startAngle parameter is a real64 value that specifies the starting angle of the pie slice. The angle is specified in degrees. Angles are measured in a counter-clockwise fashion from the vertical line going from the middle of the bounding box to the top of the bounding box (warning: this is not intuitive).

The endAngle parameter is a real64 value that specifies the ending angle of the pie slice. The angle is specified in degrees. The wPie object draws a slice of a pie graph filling in the ellipse from the startAngle to the endAngle in a counter-clockwise fashion.

The lineColor argument specifies an RGB value that HOWL uses when drawing the outline of the pie slice (the pen color). wPie objects always draw the outline with a solid line.

The fillColor argument specifies an RGB value that HOWL uses to fill the interior of the pie slice. wPie objects always fill the interior with a solid brush (color).

The bkgColor argument specifies an RGB value that HOWL uses to fill the retangular area described by x, y, w, and h that is outside the pie slice. As a general rule, this should be the same color as the background color for the form unless you want a visible rectangle surrounding the pie slice.

38.2.15 wPolygon

The wPolygon object draws a multi-vertex polygon on the screen.

```
wPolygon
(
    polyName,        // HLA identifier
    x,               // x
    y,               // y
    w,               // width
    h,               // height
    lineColor,       // linecolor (RGB)
    fillColor,       // Ellipse interior color (RGB)
    bkgColor,        // Ellipse exterior color (RGB)
    x1,              // Optional points list
    y1,              // Must have an even number of coordinates
    x2,
    y2,
    .
    .
    .
    xn,
    yn
)
```

The `polyName` argument is the identifier name that HOWL uses in the `wForm` declaration for the polygon object. This name must be unique within the `wForm` declaration.

The `x, y, w,` and `h` fields specify the coordinates and sizes for the bounding box surrounding the polygon.

The `lineColor` argument specifies an RGB value that HOWL uses when drawing the outline of the polygon (the pen color). `wPolygon` objects always draw the outline with a solid line.

The `fillColor` argument specifies an RGB value that HOWL uses to fill the interior of the polygon. `wPolygon` objects always fill the interior with a solid brush (color).

The `bkgColor` argument specifies an RGB value that HOWL uses to fill the retangular area described by `x, y, w,` and `h` that is outside the polygon. As a general rule, this should be the same color as the background color for the form unless you want a visible rectangle surrounding the polygon.

The remain arguments always appear in pairs and specify the points that make up the polygon. If you specify n points (n*2 arguments), HOWL will draw n lines between each pair of points (and between (xn,yn) and (x1,y1) to complete the closed polygon).

38.2.16 wBitmap

The wBitmap declaration creates a bitmapped object on the form.

```
wBitmap
(
    bmName,          // HLA identifier
    bmResource,      // Bitmap resource name
    x,               // x
    y,               // y
    w,               // width
    h,               // height
    bkgColor         // RGB background color
)
```

The `bmName` argument is the identifier name that HOWL uses in the `wForm` declaration for the bitmapped object. This name must be unique within the `wForm` declaration.

The `bmResource` argument is a string constant specifying the name of the bitmap resource within the executable file. Note that this is not a filename on the disk. You must use the resource editor to compile a bitmap file into the executable file. The name you provide to the resource editor for this bitmapped object is the name you use for the `bmResource` string.

The `x, y, w,` and `h` fields specify the coordinates and sizes for the bounding box surrounding the bitmap. If the `w` and `h` fields are too small, Windows will truncate the bitmap when it draws it. If the `w` and `h` fields are larger than the bitmap, Windows will fill the extra area with the value of the `bkgColor` argument.

The bkgColor argument specifies an RGB value that HOWL uses to fill the retangular area described by x, y, w, and h that is outside the bitmap. As a general rule, this should be the same color as the background color for the form unless you want a visible rectangle surrounding the bitmap.

38.2.17 wProgressBar

The wProgressBar declaration creates a progress bar object on the form.

```
wProgressBar
(
    pbName,          // HLA identifier
    x,               // x
    y,               // y
    w,               // width
    h                // height
)
```

The pbName argument is the identifier name that HOWL uses in the wForm declaration for the progress bar object. This name must be unique within the wForm declaration.

The x, y, w, and h fields specify the coordinates and sizes for the bounding box surrounding the progress bar.

See the description of the wProgressBar_t class type later in the document to learn how to specify the current progress in the progress bar.

38.2.18 wPushButton

```
wPushButton
(
  pbID,            // Identifier for push button
  caption,         // Caption for push button
  x,               // x position on form
  y,               // y position on form
  w,               // width
  h,               // height
  onClick          // "on click" event handler (or NULL)
);
```

The pbName argument is the identifier name that HOWL uses in the wForm declaration for the push button object. This name must be unique within the wForm declaration.

The caption argument is a string that Windows will display on the push button.

The x, y, w, and h fields specify the coordinates and sizes for the bounding box surrounding the push button.

The onClick argument is either NULL or the name of a widgetProc that HOWL will call when the user presses the corresponding button on the form.

38.2.19 Radio Button Objects

wForm declarations allow you to place radio buttons directly on a form or you can group a set of radio buttons together in a radio button set. In general, you'll rarely use the first form because wRadioButton and wRadioButtonLT objects don't readily exhibit button semantics. On a form by themselves, radio buttons behave just like check box objects so you're better off using check box objects than radio buttons for this purpose. Generally, wRadioButton and wRadioButtonLT objects are useful when you're building a form dynamically at run time rather than at design time with the HOWL declarative language. Nevertheless, the HOWL declarative language includes entries for wRadioButton and wRadioButtonLT objects for the sake of completeness.

38.2.19.1 wRadioButton

```
wRadioButton
(
  rbID,            // Identifier for radio button
  caption,         // Caption for radio button
```

```
    style,          // Style for radio button
    x,              // x position on form
    y,              // y position on form
    w,              // width
    h,              // height
    onClick         // "on click" event handler (or NULL)
  );
```

The `rbName` argument is the identifier name that HOWL uses in the `wForm` declaration for the radio button object. This name must be unique within the `wForm` declaration.

The `caption` argument is a string that Windows will display to the right of the radio button.

The `style` argument is either 0 (for standalone radio buttons) or one of the following constants (for radio button groups):

For the first radio button in a group:

```
w.BS_AUTORADIOBUTTON | w.WS_GROUP | w.WS_TABSTOP
```

For all but the first radio button in a group:

```
w.BS_AUTORADIOBUTTON
```

Note that you should really use the `wRadioSet` object to create sets of radio buttons rather than grouping them manually. Note that you must declare all grouped radio buttons consecutively in your source file. Any intervening widgets will end a radio set button group.

The `x, y, w,` and `h` fields specify the coordinates and sizes for the bounding box surrounding the radio button.

The `onClick` argument is either NULL or the name of a widgetProc that HOWL will call when the user presses the corresponding radio button on the form.

38.2.19.2 wRadioButtonLT

```
  wRadioButtonLT
  (
    rbID,           // Identifier for radio button
    caption,        // Caption for radio button
    style,          // Style for radio button
    x,              // x position on form
    y,              // y position on form
    w,              // width
    h,              // height
    onClick         // "on click" event handler (or NULL)
  );
```

`wRadioButtonLT` objects are identical to `wRadioButton` objects except the caption text is draw on the left side of the radio buton rather than on the right side.

38.2.19.3 wRadioSet..endwRadioSet

Functional radio buttons are created as part of a radio button set. The `wRadioSet..endwRadioSet` block encapsulates a set of `wRadioSetButton` and `wRadioSetButtonLT` objects that HOWL treats as a single set of radio buttons rather than independent buttons. The `wRadioSet` declaration takes the following form:

```
  wRadioSet
  (
    rsID,           // Identifier for radio set
    caption,        // Caption for radio set group box
    x,              // x position on form
    y,              // y position on form
    w,              // width
    h               // height
  );
```

```
    << radio set button declarations >>

  endwRadioSet
```

Only `wRadioSetButton` and `wRadioSetButtonLT` declarations may appear within a `wRadioSet..endwRadioSet` declaration and you cannot nest `wRadioSet..endwRadioSet` declarations. The `wRadioSet..endwRadioSet` declaration creates a group box with the specified bounding box. It draws the `caption` string through the line of the bounding box in the upper left hand corner.

All radio set buttons appearing in a `wRadioSet` group are treated as a single set of radio buttons. At most one radio set button will be checked in the group; pressing one button unchecks any other buttons in the same group.

Note that a `wRadioSet` object is a container object. It contains all the radio set buttons associated with the radio set.

38.2.19.3.1 wRadioSetButton

```
wRadioSetButton
(
    rsbtnID,         // HLA identifier
    caption,         // String caption for radio button
    x,               // x position
    y,               // y position
    w,               // width
    y,               // height
    onClick          // "on click" event handler
);
```

The `wRadioSetButton` declaration may only appear within a `wRadioSet..endwRadioSet` statement.

The `rsbtnID` argument must be a unique (to the form) HLA identifier. HOWL uses this identifier as the field name within the form class of the `wForm..endwForm` declaration.

The `caption` field is a string that HOWL displays to the right of the radio button image.

The `x, y, w,` and `h` fields specify the coordinates and sizes for the bounding box surrounding the radio button.

The `onClick` argument is either NULL or the name of a widgetProc that HOWL will call when the user presses the corresponding radio button on the form.

38.2.19.3.2 wRadioSetButtonLT

```
wRadioSetButtonLT
(
    rsbtnID,         // HLA identifier
    caption,         // String caption for radio button
    x,               // x position
    y,               // y position
    w,               // width
    y,               // height
    onClick          // "on click" event handler
);
```

The `wRadioSetButtonLT` declaration may only appear within a `wRadioSet..endwRadioSet` statement.

The `rsbtnID` argument must be a unique (to the form) HLA identifier. HOWL uses this identifier as the field name within the form class of the `wForm..endwForm` declaration.

The `caption` field is a string that HOWL displays to the left of the radio button image.

The `x, y, w,` and `h` fields specify the coordinates and sizes for the bounding box surrounding the radio button.

The `onClick` argument is either NULL or the name of a widgetProc that HOWL will call when the user presses the corresponding radio button on the form.

38.2.20 wRectangle

```
wRectangle
(
    rectName,        // HLA identifier
    x,               // x
    y,               // y
    w,               // width
    h,               // height
    lineColor,       // linecolor (RGB)
    fillColor        // Rectangle interior color (RGB)
)
```

The `rectName` argument is the identifier name that HOWL uses in the `wForm` declaration for the rectangle object. This name must be unique within the `wForm` declaration.

The `x, y, w,` and `h` fields specify the coordinates and sizes for the bounding box surrounding the rectangle.

The `lineColor` argument specifies an RGB value that HOWL uses when drawing the outline of the rectangle (the pen color). `wRectangle` objects always draw the outline with a solid line.

The `fillColor` argument specifies an RGB value that HOWL uses to fill the interior of the rectangle. `wRectangle` objects always fill the interior with a solid brush (color).

Note that there is no background color (as exists for other graphic objects). This is because the rectangle completely fills the bounding box so there is no need to fill the background area as it never shows through.

38.2.21 wRoundRect

```
wRoundRect
(
    rrectName,       // HLA identifier
    x,               // x
    y,               // y
    w,               // width
    h,               // height
    cw,              // Corner width
    cht,             // Corner height
    lineColor,       // linecolor (RGB)
    fillColor,       // Round rectangle interior color (RGB)
    BkgColor         // Round rectangle exterior color (RGB)
)
```

The r`rectName` argument is the identifier name that HOWL uses in the `wForm` declaration for the round rectangle object. This name must be unique within the `wForm` declaration.

The `x, y, w,` and `h` fields specify the coordinates and sizes for the bounding box surrounding the round rectangle.

The `cw` and `cht` arguments specify the width and height of the ellipse used to draw the corners of the round rectangle.

The `lineColor` argument specifies an RGB value that HOWL uses when drawing the outline of the round rectangle (the pen color). `wRoundRect` objects always draw the outline with a solid line.

The `fillColor` argument specifies an RGB value that HOWL uses to fill the interior of the rectangle. `wRectangle` objects always fill the interior with a solid brush (color).

Note that there is no background color (as exists for other graphic objects). This is because the rectangle completely fills the bounding box so there is no need to fill the background area as it never shows through.

38.2.22 wScrollBar

```
wScrollBar
(
    scrollBarID,     // Scrollbar name (HLA id)
```

```
        x,              // x
        y,              // y
        w,              // width
        h,              // height
        style,          // Scroll bar style
        onChange        // On change handler
    )
```

The `scrollBarID` argument is the identifier name that HOWL uses in the `wForm` declaration for the scroll bar object. This name must be unique within the `wForm` declaration.

The `x, y, w,` and `h` fields specify the coordinates and sizes for the bounding box surrounding the scroll bar.

The `style` argument specifies the scroll bar style and is the logical-OR of zero or more of the following constants:

w.SBS_BOTTOMALIGN	Aligns the bottom edge of the scroll bar with the bottom edge of the rectangle defined by the CreateWindow parameters x, y, nWidth, and nHeight. The scroll bar has the default height for system scroll bars. Use this style with the SBS_HORZ style.
w.SBS_HORZ	Designates a horizontal scroll bar. If neither the SBS_BOTTOMALIGN nor SBS_TOPALIGN style is specified, the scroll bar has the height, width, and position specified by the parameters of CreateWindow.
w.SBS_LEFTALIGN	Aligns the left edge of the scroll bar with the left edge of the rectangle defined by the parameters of CreateWindow. The scroll bar has the default width for system scroll bars. Use this style with the SBS_VERT style.
w.SBS_RIGHTALIGN	Aligns the right edge of the scroll bar with the right edge of the rectangle defined by the parameters of CreateWindow. The scroll bar has the default width for system scroll bars. Use this style with the SBS_VERT style.
w.SBS_SIZEBOX	Designates a size box. If you specify neither the SBS_SIZEBOXBOTTOMRIGHTALIGN nor the SBS_SIZEBOXTOPLEFTALIGN style, the size box has the height, width, and position specified by the parameters of CreateWindow.
w.SBS_SIZEBOXBOTTOMRIGHTALIGN	Aligns the lower-right corner of the size box with the lower-right corner of the rectangle specified by the parameters of CreateWindow. The size box has the default size for system size boxes. Use this style with the SBS_SIZEBOX style.
w. SBS_SIZEBOXTOPLEFTALIGN	Aligns the upper-left corner of the size box with the upper-left corner of the rectangle specified by the parameters of CreateWindow. The size box has the default size for system size boxes. Use this style with the SBS_SIZEBOX style.
w.SBS_SIZEGRIP	Same as SBS_SIZEBOX, but with a raised edge.
w.SBS_TOPALIGN	Aligns the top edge of the scroll bar with the top edge of the rectangle defined by the parameters of CreateWindow. The scroll bar has the default height for system scroll bars. Use this style with the SBS_HORZ style.
w.SBS_VERT	Designates a vertical scroll bar. If you specify neither the SBS_RIGHTALIGN nor the SBS_LEFTALIGN style, the scroll bar has the height, width, and position specified by the parameters of CreateWindow.

The `onChange` argument is the name of a `widgetProc` that HOWL will call whenever there is a change made to the scroll bar's position.

38.2.23 wTextEdit

A wTextEdit object allows the user to enter a text file object.

```
wTextEdit
```

```
(
    teName,       // HLA identifier for this object
    InitialText,  // Initial text for text edit object
    x,            // x position
    y,            // y position
    w,            // width
    h,            // height
    s,            // style
    onChange      // onChange handler (can be NULL)
);
```

The `teName` argument is the name that HOWL will use in the `wForm` declaration for this `wTextEdit` object. This name must be unique within the `wForm` declaration.

The `InitialText` argument is a string (usually empty) that HOWL uses to initialize the text editor's text field when the form is first created.

The `x, y, w,` and `h` arguments specify the position and size of the `wTextEdit` object on the form.

The `s` argument is the Windows editbox style that HOWL logically ORs with the value `(w.ES_MULTILINE | w.ES_WANTRETURN | w.WS_HSCROLL | w.WS_VSCROLL)`. See the discussion of the legal values in the section on `wEditBox`.

The `onChange` argument is the name of a `widgetProc` that HOWL will call whenever the user changes any text in the text editor. This field can be NULL, in which case HOWL won't bother to call any procedure when the user changes the text (this is actually the most common situation; usually the program will determine the currently selected item in a `wTextEdit` when some other event occurs, and ignore the immediate changes that might occur in a `wTextEdit` object). Note that if onChange is non-NULL, then HOWL will call the `widgetProc` any time there is a single-character change to the text edit object; this is probably more often than you'd like, which is why this field is generally NULL and applications simply read the data from the text edit object when some other event occurs.

38.2.24 wTrackBar

```
wTrackBar
(
    trackBarID,      // Trackbar name (HLA id)
    x,               // x
    y,               // y
    w,               // width
    h,               // height
    style,           // Track bar style
    onChange         // On change handler
)
```

The `trackBarID` argument is the identifier name that HOWL uses in the `wForm` declaration for the track bar object. This name must be unique within the `wForm` declaration.

The `x, y, w,` and `h` fields specify the coordinates and sizes for the bounding box surrounding the track bar.

The `style` argument specifies the track bar style and is the logical-OR of zero or more of the following constants:

w.TBS_HORZ	Designates a horizontal track bar (this is the default).
w.TBS_TOP	Display tick marks on the top of a horizontal track bar.
w.TBS_BOTTOM	Display tick marks on the bottom of a horizontal track bar (default).
w.TBS_VERT	Designates a vertical track bar.
w.TBS_LEFT	Display tick marks on the left side of a vertical track bar.
w.TBS_RIGHT	Display tick marks on the right side of a vertical track bar (default).
w.TBS_BOTH	Display tick marks on both sides of a track bar (vert or horz).

Note that all `wTrackBar` objects have the w.TBS_AUTOTICKS style.

The `onChange` argument is the name of a `widgetProc` that HOWL will call whenever there is a change made to the scroll bar's position.

38.2.25 wUpDown

```
wUpDown
(
    upDownID,        // Up/down control object name
    style,           // No special format/style/alignment
    x,               // x
    y,               // y
    w,               // width
    h,               // height
    min,             // Minimum position
    max,             // Maximum position
    initial,         // Initial position
    onUpDown         // Click event handler
);
```

A `wUpDown` widget is a pair of arrow buttons that the user can click on to increment or decrement a value. `wUpDown` objects are stand-alone (see wUpDownEditBox for a version that is connected to an edit box).

The `upDownID` field is an HLA identifier that HOWL uses as the name of the object on the form. This name must be unique within the form class declaration.

The `style` argument is one of the following values:

`UDS_ALIGNLEFT` — Positions the up-down control next to the left edge of the buddy window. The buddy window is moved to the right and its width decreased to accommodate the width of the up-down control. This style is generally used only with the `UDS_AUTOBUDDY` style. See `wUpDownEditBox` for additional details concerning buddy controls.

`UDS_ALIGNRIGHT` — Positions the up-down control next to the right edge of the buddy window. The width of the buddy window is decreased to accommodate the width of the up-down control. This style is generally used only with the `UDS_AUTOBUDDY` style. See `wUpDownEditBox` for additional details concerning buddy controls.

`UDS_ARROWKEYS` — Causes the up-down control to increment and decrement the position when the UP ARROW and DOWN ARROW keys are pressed.

`UDS_AUTOBUDDY` — Automatically selects the previous window in the Z order as the up-down control's buddy window.

`UDS_HORZ` — Causes the up-down control's arrows to point left and right instead of up and down.

`UDS_NOTHOUSANDS` — Does not insert a thousands separator between every three decimal digits.

`UDS_SETBUDDYINT` — Causes the up-down control to set the text of the buddy window (using the WM_SETTEXT message) when the position changes. The text consists of the position formatted as a decimal or hexadecimal string.

`UDS_WRAP` — Causes the position to "wrap" if it is incremented or decremented beyond the ending or beginning of the range.

The `x, y, w,` and `h` arguments specify the position and size of the `wUpDown` object on the form. HOWL ignores these values if you specify a non-NULL buddy value; in that case, HOWL uses the bounding box of the `wEditBox` object to control the placement of the up/down arrows.

The `min` argument specifies the minimum value that a `wUpDown` object will return. If the control's current value is equal to the `min` value and the user presses the down arrow, the `wUpDown` object will not decrement the value.

The max argument specifies the maximum value that a wUpDown object will return. If the control's current value is equal to the max value and the user presses the up arrow, the wUpDown object will not increment the value.

The initial argument specifies the initial value of the wUpDown object when the form is created.

The onUpDown argument is either NULL or specifies the name of a widgetProc procedure that HOWL will call whenever the user presses an up or down arrow on the control. If this field is NULL, then HOWL will not call a function whenever an up or down arrow is pressed and the application will have to call an appropriate wUpDown method to retrieve the current value of the wUpDown control.

38.2.26 wUpDownEditBox

```
wUpDownEditBox
(
    upDownID,        // Up/down control object name
    style,           // No special format/style/alignment
    x,               // x
    y,               // y
    w,               // width
    h,               // height
    min,             // Minimum position
    max,             // Maximum position
    initial,         // Initial position
    onTextChange,    // On Change event handler (edit box)
    onUpDown         // Click event handler (up/down arrow)
);
```

A wUpDown widget is a pair of arrow buttons that the user can click on to increment or decrement a value. wUpDown objects can be stand-alone or they can be associated with a wEditBox object (the "buddy"). When a wUpDown object is associated with a buddy wEditBox object, the arrows are connect to the edit box and clicking on the up or down arrows produces a string in the wEditBox object representing the current value of the wUpDown object.

The style argument is one of the following values:

UDS_ALIGNLEFT	Positions the up-down control next to the left edge of the buddy window. The buddy window is moved to the right and its width decreased to accommodate the width of the up-down control.
UDS_ALIGNRIGHT	Positions the up-down control next to the right edge of the buddy window. The width of the buddy window is decreased to accommodate the width of the up-down control.
UDS_ARROWKEYS	Causes the up-down control to increment and decrement the position when the UP ARROW and DOWN ARROW keys are pressed.
UDS_AUTOBUDDY	Automatically selects the previous window in the Z order as the up-down control's buddy window. This style shouldn't be used with wUpDownEditBox objects.
UDS_HORZ	Causes the up-down control's arrows to point left and right instead of up and down.
UDS_NOTHOUSANDS	Does not insert a thousands separator between every three decimal digits.
UDS_SETBUDDYINT	Causes the up-down control to set the text of the buddy window (using the WM_SETTEXT message) when the position changes. The text consists of the position formatted as a decimal or hexadecimal string.
UDS_WRAP	Causes the position to "wrap" if it is incremented or decremented beyond the ending or beginning of the range.

The x, y, w, and h arguments specify the position and size of the wUpDown object on the form. HOWL ignores these values if you specify a non-NULL buddy value; in that case, HOWL uses the bounding box of the wEditBox object to control the placement of the up/down arrows.

The min argument specifies the minimum value that a wUpDown object will return. If the control's current value is equal to the min value and the user presses the down arrow, the wUpDown object will not decrement the value.

The max argument specifies the maximum value that a wUpDown object will return. If the control's current value is equal to the max value and the user presses the up arrow, the wUpDown object will not increment the value.

The initial argument specifies the initial value of the wUpDown object when the form is created.

The onTextChange argument is either NULL or specifies the name of a widgetProc procedure that HOWL will call whenever the user changes a value in the edit box control. If this field is NULL, then HOWL will not call a function whenever the edit box changes and the application will have to call an appropriate wUpDownEditBox method to retrieve the current value of the wUpDownEditBox's edit box control. Note that pressing an up or down error will cause a change to the edit box, which will call HOWL to call this function.

The onUpDown argument is either NULL or specifies the name of a widgetProc procedure that HOWL will call whenever the user presses an up or down arrow on the control. If this field is NULL, then HOWL will not call a function whenever an up or down arrow is pressed and the application will have to call an appropriate wUpDown method to retrieve the current value of the wUpDown control.

38.2.27 wTimer

```
wTimer
(
    timerID,          // Timer control object name
    period,           // Timeout value in milliseconds
    timing,           // Type of timer (wTimer_t.oneShot or wTimer_t.periodic)
    onTimeOut         // Timeout event handler
);
```

A wTimer widget creates a small background thread that calls the onTimeOut widget after some period of time. Timers operate on one of two modes: oneShot or periodic. In the wTimer_t.oneShot mode , the timer delays for at least the number of milliseconds specified by the period argument and then calls the onTimeOut widgetProc exactly once. In the wTimer_t.periodic mode, the timer calls the onTimeOut widgetProc once every period milliseconds.

Note that declaring a wTimer object in the HOWL declarative language does not actually start the timer operating. It initializes the object, but you must call the wTimer_t.start method associated with the class to actually begin the timing process. See the description of the wTimer_t class later in this document for more details.

38.2.28 wWindow..endwWindow

A wWindow object is a container. It specifies a rectangular area on a form that contains other objects. All the widgets you declare between a wWindow statement and the corresponding endwWindow terminator will be contained by the window (and can be treated as a whole) at run time.

```
wWindow
(
    windowID,         // HLA identifier
    caption,          // Window title (ignored unless style calls for title)
    exStyle,          // Extended style for window
    style,            // Windows' style for window
    x,                // x position
    y,                // y position
    w,                // width
    h,                // height
    bkgColor          // RGB background color for window
)

    <<Other widget declarations >>

endwWindow
```

38.3 The HOWL Run-time Library

Although the HOWL declarative language (the `wForm..endwForm` macro) makes it very easy to design forms, your applications will need to interact with the HOWL run-time library code in order to make full use of HOWL's capabilities. This section of this document describes the run-time semantics of the HOWL library.

The first thing to note is that HOWL is an object-oriented library. Almost all HOWL data types and code are part of the HOWL object hierarchy. So the best place to start when describing HOWL is with a description of the object hierarchy.

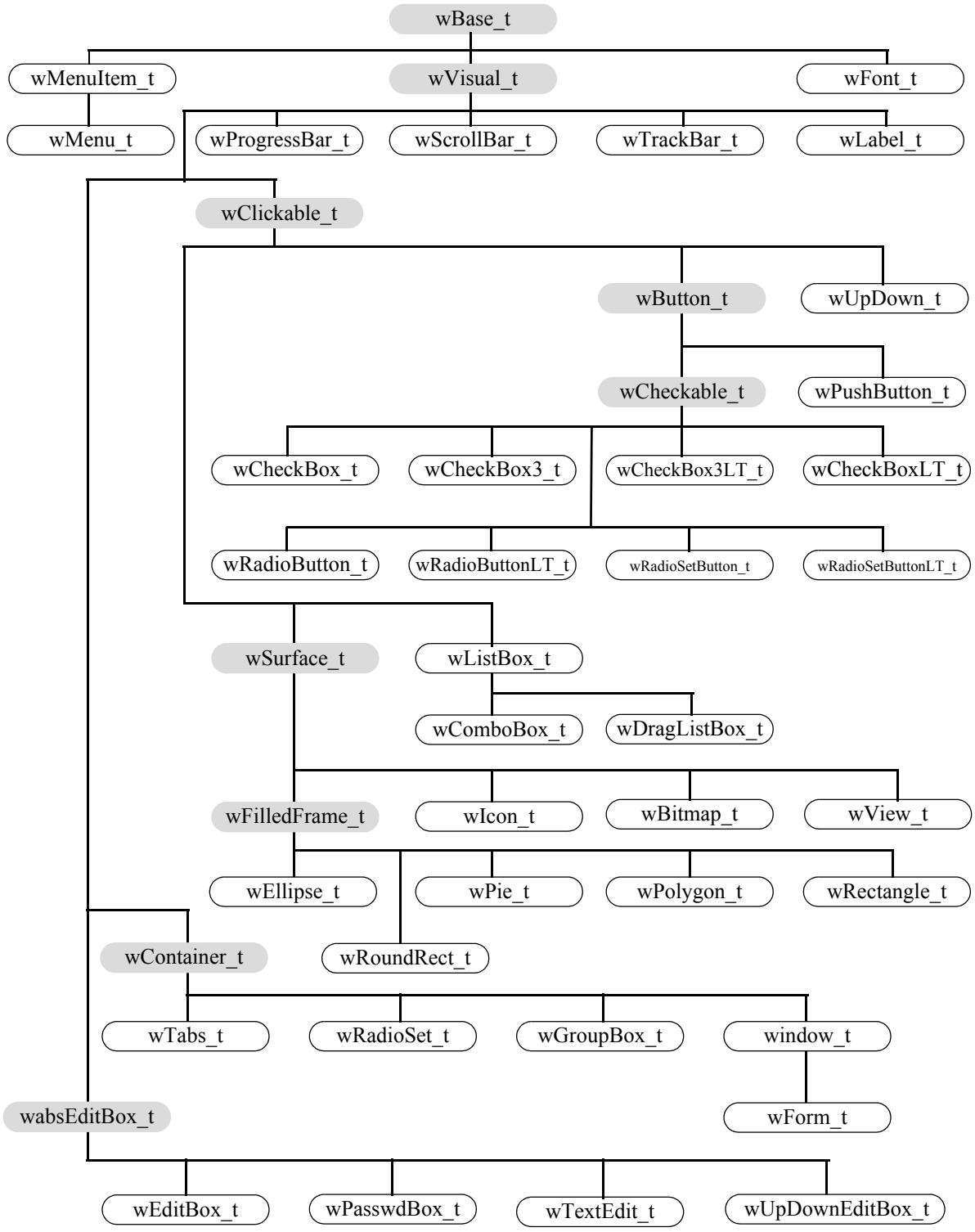
HOWL contains a single base class, appropriately named `wBase_t`. All other objects in HOWL are derived from this class. We'll describe `wBase_t` completely in the next section, but the important thing to note is that `wBase_t` is the root of the class hierarchy tree for HOWL.

As you should know from object-oriented programming, descendant (child/derived) classes inherit all the fields of their base (parent/ancestor) classes. Therefore, all the classes in the HOWL object hierarchy inherit the fields of the `wBase_t` class (and all other ancestor classes to that particular class). In the following sections that describe each of the classes in the HOWL hierarchy, the descriptions will only discuss the fields that are specific to a given class; this document assumes that you understand that each class will inherit fields from all the ancestor classes and that you should look at the documentation for those ancestor classes in order to get the full picture for each class.

Every HOWL object (that is derived from `wBase_t`) contains a special `wType` field. This is an lword (128-bit) object that HOWL uses to maintain run-time type information about that particular object. This is an array of 128 bits that specify membership/absence from a particular class. When an application is given a generic pointer to an object of any HOWL type (e.g., `wBase_t`), the application can test this array to see if that object is a specific type (or is derived from a specific type). To accommodate this, HOWL defines a set of constants for each of that HOWL class types (except `wBase_t`) that have the following names and functions:

- *typename*_b ("b" stands for "bit number") is a small integer number between zero and the number of HOWL class types (less that 128) that associates a unique enumerated value with each HOWL class type. This also provides an index into the `wType` bit array.
- *typename*_ps ("ps" stands for "power set") is a singleton set constant containing a "1" bit at index *typename*_b with all other bits containing zero.
- *typename*_c ("c" stands for "constant") is a constant (up to 128 bits) with a "1" bit in each bit position specifying whether typename is a descendant (or is) the type indicated by the *typename*_b bit position into *typename*_c. For example, `wProgressBar_c` would contain set bits in bit positions `wProgressBar_b` and `wVisual_p` because `wProgressBar_t` is derived from `wVisual_t`. Because all HOWL objects are derived from `wBase_t`, there is no need to set aside a bit position for `wBase_t` in `wProgressBar_c`. Note that *typename*_c is generated via the logical-OR of *typename*_ps and all the ancestor class "_ps" values for *typename*_t.

The following diagram shows the HOWL object hierarchy. The nodes in gray are abstract classes; you do not normally create objects of these types (generally, you only create objects of types derived from abstract base classes).


wBase_t
wMenuItem_t
wVisual_t
wFont_t
wMenu_t
wProgressBar_t
wScrollBar_t
wTrackBar_t
wLabel_t
wClickable_t
wButton_t
wUpDown_t
wCheckable_t
wPushButton_t
wCheckBox_t
wCheckBox3_t
wCheckBox3LT_t
wCheckBoxLT_t
wRadioButton_t
wRadioButtonLT_t
wRadioSetButton_t
wRadioSetButtonLT_t
wSurface_t
wListBox_t
wComboBox_t
wDragListBox_t
wFilledFrame_t
wIcon_t
wBitmap_t
wView_t
wEllipse_t
wPie_t
wPolygon_t
wRectangle_t
wContainer_t
wRoundRect_t
wTabs_t
wRadioSet_t
wGroupBox_t
window_t
wabsEditBox_t
wForm_t
wEditBox_t
wPasswdBox_t
wTextEdit_t
wUpDownEditBox_t

38.3.1 Private Data Fields

Many HOWL classes contain private data fields. Although HLA will not prevent you from accessing these private data fields, application programmers should avoid direct access of these private fields. Access to the private data fields is intended for use by HOWL functions only.

For those data fields whose values might be of interest to a HOWL application programmer, the HOWL library generally provides accessor ("getter") and mutator ("setter") functions that let you access these private fields. You should always attempt to use these accessor/mutator functions for all private data access. Reading the values of some private fields (by calling the accessor functions) may cause HOWL to make a Win32 API call to make the value consistent with Windows; writing to a private data field (via a mutator) may cause HOWL to execute some additional code to tell Windows (or the rest of HOWL) about the change. Reading and writing these private data fields directly may circumvent these actions that keep HOWL's internal data structures consistent.

The private fields in a class are easily distinguished in the howl.hhf header file; the header collects all private data fields into record variables within the classes that have a "_private" suffix. Unless you are writing a class that is an extension of the HOWL library, you should not directly access these fields.

38.3.2 Abstract Classes

The HOWL class hierarchy contains several abstract base classes that combine features common to various concrete classes. The following subsections describe each of these base classes.

38.3.2.1 wBase_t

The `wBase_t` class is the root class of the entire HOWL hierarchy. This class has the following definition:

```
wBase_t:
    class

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

            wBase_private:
                record

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

                    objectID        :dword;
                    nextWidget      :wBase_p;

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

                    parentForm      :wForm_p;

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

                    parentHandle    :dword;
```

```
                endrecord;


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

            // Constructors/Destructors:

            procedure create_wBase
            (
                wbName  :string
            );  external;

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

            // Accessor/mutator functions:

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

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

            // Default message processor:

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

        endclass;
```

`objectID_g` This is a static field that HOWL uses to dynamically assign unique Windows identifiers to objects. Applications should not reference this field; they must not modify the value of this field. HOWL automatically increments this field whenever you create an instance of some HOWL object.

`wType` The wType field is a 128-bit bit array that provides run-time type information about an object to the application. If you test bit position *typename*_b in the wType field, you can determine if the current object is derived from (or is) type *typename*. The *howl.hhf* header file defines *typename*_b constants for all the HOWL types (subsituting the appropriate type name, such as `wButton_t`, for *typename*). This is a public field for read-only access. Applications must never modify its value.

`handle` Almost all HOWL objects have a Windows handle associated with them. The `handle` field contains this value. Technically, handle ought to be a private data field (there is even an accessor function for it), however, because applications need to frequently access this

field, it was made public. Note, however, that an application should never write data to this field.

`_name` This field is a string representation of the object's name in the main form. This field is mainly useful for testing, debugging, and tracing purposes. Other than initializing this string pointer, the HOWL library doesn't access this field at all, so an application is free to use this field however it wants.

`visible` This boolean variable contains true if the object is a `wVisual_t` object and is visible on the form. It contains false if the object is not visible. If the object isn't a `wVisual_t` (or descendant) object, then this field's value is meaningless. This is a private field, always use the accessor function to retrieve its value. Applications must never directly change the value of this field.

`enabled` This boolean variable contains true if the object is a `wVisual_t` object and is enabled on the form. It contains false if the object is not enabled. If the object isn't a `wVisual_t` (or descendant) object, then this field's value is meaningless. This is a private field, always use the accessor function to retrieve its value. Applications must never directly change the value of this field.

`onHeap` This field contains true if the object's storage is allocated on the heap. It contains false if the object's storage is not allocated on the heap. If you initialize an object in storage that is not on the heap, it is your responsibility to set this field to false. If you create an object and request heap allocation for it (by calling a class procedure constructor with ESI equal to zero), the HOWL constructors will automatically set this field to true. This is a private field, always use the accessor function to retrieve its value. Applications must never directly change the value of this field.

`objectID` The `objectID` field contains the specific Windows ID (if applicable) for the current object. The create method for an object generally copies the global `objectID_g` value to this field and then increments the global value to generate unique ID values for each object. For the most part, HOWL ignores this field (it identifies objects by the pointer to the object rather than by the Windows ID). This is a private field, always use the accessor function to retrieve its value. Applications must never directly change the value of this field.

`nextWidget` `wContainer_t` objects use this field to create a linked list of widgets contained by the container object. All objects created via the HOWL declarative language (except the wForm object) are contained by some object (e.g., the wForm object). However, it is possible to dynamically instantiate objects that are not contained by a form, and the form object itself isn't contained by another container, so you cannot assume that this field contains a valid value unless you iterator across the widgets of a container object. This is a private data field, no application program access is legal.

`parentForm` This is a pointer to the wForm object that holds the current widget. Note that this is the actual object pointer, not the form's handle. This is a private field, always use the accessor/mutator functions to read/write its value.

`parentHandle` This is the window handle of the Windows' object on which the current widget is a child control. This is a private field, always use the accessor/mutator functions to read or write its value.

`create_wBase` The `create_wBase` procedure is the constructor for the class. Because `wBase_t` is an abstract class, you never instantiate objects of type `wBase_t`. Unless you are writing a constructor for a new class you've derived from `wBase_t`, you will probably never call this constructor. This constructor is responsible for setting up the object's _name field (passed as an argument), setting up the ObjectID field, and initializing all the other fields to reasonable default values (that the derived classes' constructors will probably overwrite).

`destroy` This is the base level destructor function. You do not generally call this method directly; instead, a higher-level destructor function will probably call this function when you

invoke destroy on some object. The `wBase_t.destroy` method checks the `onHeap` field and will deallocate the storage associated with the object if the storage was allocated on the heap.

`show`,

`hide` These two methods simply store true (`show`) or false (`hide`) into the `visible` field. Other than that, they do nothing. They are included in `wBase_t` just to allow code to show and hide all objects. Generally, `wBase_t` descendant classes (usually `wVisual_t` descendants) override these methods to show or hide a visual object on the screen.

`enable`,

`disable` These two methods store true (`enable`) or false (`disable`) into the `enabled` field.. Other than that, they do nothing. They are included in `wBase_t` just to allow code to enable and disable objects. Generally, `wBase_t` descendant classes (usually `wVisual_t` descendants) override these methods to enable or disable a visual object on the screen.

`get_handle`,

`get_objectID`,

`get_visible`,

`get_enabled`,

`get_onHeap`,

`get_parentHandle`

`get_parentForm` These are "accessor" functions that retrieve the value of the associated class field. As a general rule you should always call the accessor function to retrieve an object's data field values as the accessor might contain code to "condition" those values prior to consumption

`set_onHeap` This is a "mutator" function that lets an application write a value to the `onHeap` field. Applications should always call this mutator rather than writing directly to the `onHeap` field because their might be code in the mutator that does additional processing required by the class.

`set_parentHandle`

This is a "mutator" function that lets an application write a value to the `parentHandle` field. Applications should be very careful about writing to this field. The only time an application should write to `parentHandle` is when it dynamically creates a new object at run-time (or, in the rare case of moving a widget from one form to another).

`set_parentForm` This is a "mutator" function that lets an application write a value to the `parentForm` field. Applications should be very careful about writing to this field. The only time an application should write to `parentHandle` is when it dynamically creates a new object at run-time (or, in the rare case of moving a widget from one form to another).

`processMessage` The `processMessage` method is used by HOWL to do default Windows message processing when no other object handles a message sent from Windows. This is a Windows callback function and you should never call it directly unless you're extending HOWL by added new classes (and you wind up calling this code from the `processMessage` function in your new class). See the HOWL source code for more details on this function.

38.3.2.2 wVisual_t

The wVisual_t class, derived from wBase_t, contains the basic information that all visual objects (that is, those appearing on a form) possess. Of course, as wVisual_t is derived from wBase_t, all wVisual_t objects include all the fields from the wBase_t type.

```
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;
```

`x, y, width,`

`height` These fields form a bounding rectangle into which the object will appear. These are private data fields. HOWL applications should use the accessor/mutator functions to change their values.

`bkgColor` This field holds an RGB value representing the background color for the window defined by the `wVisual_t` object. Never access this field directly; always use the accessor/mutator functions so that HOWL can properly update the private `bkgBrush` field. Note all objects derived from the wVisual_t class use this field,those objects that do not simply ignore its value.

`bkgBrush` This is the brush that Windows uses to paint the background color for the `wVisual_t` object. Note that this is a private field and applications should never access it. HOWL computes the value for this field from the `bkgColor` field. Note all objects derived from the wVisual_t class use this field,those objects that do not simply ignore its value.

`style` This is the window style used for various window objects. Not all wVisual_t objects make use of this field.

`exStyle` This is the window extended style used for various window objects. Not all wVisual_t objects make use of this field.

`create_wVisual` This is the constructor for the class. Because this is an abstract base class, you must never call this function with ESI containing NULL. In general, you will never directly call this function unless you are creating your own HOWL classes. Whenever you call the constructor for a concrete HOWL class, it will automatically call this constructor for you.

`get_x,`
`get_y,`
`get_width,`
`get_height,`
`get_bkgColor`
`get_style,`
`get_exStyle`

These are the "accessor" functions for this class. You should call these functions to retrieve any data fields for the object.

`set_x,`
`set_y,`
`set_width,`
`set_height,`
`set_bkgColor`
`set_style`
`set_exStyle` These are the "mutator" functions for the class. You must call these functions rather than storing values directly into the data fields for the object. These function do additional work that is necessary for the system (such as redrawing objects when you change their possition or size).

`move` This is a combination of set_x and set_y rolled into a single convenient package (which causes less redraw flashing on the screen versus making the two separate calls).

`resize` This is a combination of set_width and set_height rolled into a single convenient package (which causes less redraw flashing on the screen versus making the two separate calls).

`show, hide`	These methods make a visual object visible (`show`) or invisible (`hide`) on the form. These methods are also responsible for updating the object's `visible` field.
`enable, disable`	These methods enable or disable (gray) an object on the form. Note that not all visual objects can be enabled or disabled. Those that cannot be disabled simply ignore calls to these methods.
`set_focus`	This method changes the window focus to the current object.
`onCreate,` `onClose`	These methods are intended for internal use by the HOWL system. You should never call them directly.

38.3.2.3 wClickable_t

The wClickable_t type is an abstract base class derived from wVisual_t that contains fields and code associated with objects that the user can click on (with the mouse) on the form. This class inherits all the fields from wVisual_t. This class handles both single-click and double-click events. Some objects don't support double-clicking, in which case the double-click facilities wind up being unused.[4]

```
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;
```

4. Technically, this class should have been split into two classes: wSingleClickable_t and wDoubleClickable_t (derived from wSingleClickable_t) and derived classes that don't support double-clicking would simply be derived from wSingleClickable_t. However, it's probably a bit more efficient to implement the single class and ignore double-click operations if they aren't used.

onClick	This is a pointer to a widgetProc procedure that HOWL will call when the user clicks on a wClickable_t object. This field must either contain NULL (meaning HOWL will ignore the click operation) or the address of a widgetProc procedure.
onDblClick	This is a pointer to a widgetProc procedure that HOWL will call when the user double-clicks on a wClickable_t object. If this field contains NULL, then HOWL disables the double-click operation. Note that not all wClickable_t objects support double-clicking, so this field may be ignored by HOWL. Also note that if the user double-clicks on an object (that supports double clicking) and both the onClick and onDblClick fields contain non-NULL values, HOWL will call the onClick widgetProc procedure twice and the onDblClick widgetProc procedure once.

create_wClickable	This is the class constructor. Applications will not normally call this procedure (the constructors for derived classes will call this procedure).
get_onClick, get_onDblClick	These methods return the value of the onClick and onDblClick data fields. Application programs should always call these "accessor" functions rather than directly accessing these fields.
set_onClick, set_onDblClick	These "mutator" functions set the value of the onClick and onDblClick data fields.
click	This method simulates a click on the current object's button.

38.3.2.4 wButton_t

The wButton_t type is an abstract class that contains common fields for all the button, checkbox, and radio button class types. This class is derived from wClickable_t, so it inherits all the wClickable_t fields.

```
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;
```

`onPaint` This is a pointer to a `widgetProc` procedure that HOWL will call whenever the button is painted on the screen. If this pointer contains NULL, HOWL does not call any user-defined `onPaint` procedure. On older versions of Windows this notification was used for owner-drawn buttons. Newer versions of Windows use the "owner drawn" style for this purpose. By default, the `wButton_t.create_wButton` constructor initializes this field with NULL.

`onHilite` This is a pointer to a `widgetProc` that HOWL will call (if the pointer is non-NULL) when a button is first pressed. On older versions of Windows, this was used to draw the button in a special depressed state. In HOWL, you can use this event to trigger some operation when the button is first clicked. By default, the `wButton_t.create_wButton` constructor initializes this field with NULL.

`onUnHilite` This is a pointer to a `widgetProc` that HOWL will call (if the pointer is non-NULL) when a button is released. On older versions of Windows, this was used to draw the button in a normal non-depressed state. In HOWL, you can use this event to trigger some operation when the button is released. By default, the `wButton_t.create_wButton` constructor initializes this field with NULL.

`onDisable` This is a pointer to a `widgetProc` that HOWL will call (if the pointer is non-NULL) when a button is disabled. On older versions of Windows, this was used to draw the button in a disabled state. In HOWL, you can use this event to trigger some operation when the button is disabled. By default, the `wButton_t.create_wButton` constructor initializes this field with NULL.

`onSetFocus` This is a pointer to a `widgetProc` that HOWL will call (if the pointer is non-NULL) when focus is shifted to the button. By default, the `wButton_t.create_wButton` constructor initializes this field with NULL.

`onKillFocus` This is a pointer to a `widgetProc` that HOWL will call (if the pointer is non-NULL) when focus is shifted away from the button. By default, the `wButton_t.create_wButton` constructor initializes this field with NULL.

Note that `wButton_t` objects inherit the remaining button notification functions, `onClick` and `onDblClick`, from the `wClickable_t` class.

`create_wButton`	The `create_wButton` procedure is the constructor for this class. Like all the constructors in HOWL abstract classes, user application do not normally call this constructor directly; constructors in derived classes will call this procedure.

`get_onPaint,`
`get_onHilite,`
`get_onUnHilite,`
`get_onDisable,`
`get_onSetFocus,`
`get_onKillFocus`
	These are accessor functions that return the values of the corresponding widgetProc function pointers. Note that the `create_wButton` construction initializes all these pointers to NULL when an object is first created.

`set_onPaint,`
`set_onHilite,`
`set_onUnHilite,`
`set_onDisable,`
`set_onSetFocus,`
`set_onKillFocus`
	These are the mutator functions that let you set the addresses of the event handler functions for this class.

`get_text`	This function retrieves the caption text for a button and stores that text into the string passed as a parameter to this function. The string you pass to this function must have sufficient storage allocated for it to hold the caption or this function will raise an ex.StringOverflow exception.
`a_get_text`	This function makes a copy of the button's caption on the heap and returns a pointer to this string in the EAX register. It is the caller's responsibility to free the storage associated with this string when it is done using the string data.
`set_text`	This function changes the button's caption text to the string value you pass as an argument.
processMessage	This is an internal HOWL function. User applications do not call this method.

38.3.2.5 wCheckable_t

The `wCheckable_t` class is an abstract base class for button objects that are "checkable". This includes the various check boxes and radio buttons.

```
wCheckable_t:
    class inherits( wButton_t );

        procedure create_wCheckable_t
        (
            wchkName    :string;
            caption     :string;
            style       :dword;
            parent      :dword;
            x           :dword;
            y           :dword;
            width       :dword;
            height      :dword;
            onClick     :widgetProc
        );  external;

        method set_check( state:dword );                external;
        method get_check; @returns( "eax" );            external;
```

```
    endclass;
```

`create_wCheckable`
: The procedure is the constructor for this class. Like all abstract base class constructors, applications should not call this procedure. The constructors for concrete classes will make calls to this constructor as appropriate.

`get_check`
: This method retrieves the current state of the checkable button. It returns true in EAX if the button is checked, false if the button is not checked.

`set_check`
: This method sets the current state of the checkable button. If the single argument contains true, the button will be checked; if the argument contains false, the button will be unchecked.

38.3.2.6 wSurface_t

The `wSurface_t` abstract base class represents a single "window" on a form onto which HOWL can draw things. This abstract class, for example, is the base class for graphic objects like rectangles as well as HOWL views and windows. `wSurface_t` objects are clickable (as this class inherits the `wClickable_t` class).

```
wSurface_t:
    class inherits( wClickable_t );

        var
            align( 4 );
            wSurface_private:
                record

                    // onPaint event pointer:

                    onPaint     :widgetProc;

                endrecord;

        procedure create_wSurface
        (
            wsName      :string;
            exStyle     :dword;
            style       :dword;
            parent      :dword;
            x           :dword;
            y           :dword;
            width       :dword;
            height      :dword;
            bkgColor    :dword;
            visible     :boolean
        );  external;

        override method destroy;                        external;
        override method processMessage;                 external;
        override method onClose;                        external;
        override method onCreate;                       external;

        method get_onPaint;     @returns( "eax" );      external;
        method set_onPaint( onPaint:widgetProc );       external;

    endclass;
```

`onPaint`	This widgetProc pointer is either NULL or points at a procedure that HOWL will call when it receives a w.WM_PAINT message for the surface.
`create_wSurface`	The `create_wSurface` procedure is the constructor for the wSurface_t class. Like all abstract base class constructors, applications will not directly call this procedure -- the derived class constructors are the ones that will call this procedure.
`destroy`	The `destroy` method is responsible for freeing up the storage associated with the object and the `_bkgBrush` system resource when an application is done using a `wSurface_t` object. Normally, applications will not directly call this destructor. Instead, derived class destructor methods will call this method when they are destroyed.
`processMessage`, `onClose`, `onCreate`	These are private methods used by HOWL. Application programs should not call these methods.
`get_onPaint`, `set_onPaint`	These are the accessor/mutator functions that get/set the address of the onPaint event-handling widgetProc.

38.3.2.7 wFilledFrame_t

The `wFilledFrame_t` abstract base class is used for objects that contain graphic entities drawn with a line and filled with an interior color. This includes objects such as rectangles, ellipses, and round rectangles. This extends the `wSurface_t` type by adding a line drawing color and a fill color (on top of the background color provided by `wSurface_t`).

```
wFilledFrame_t:
    class inherits( wSurface_t );
        var
            align( 4 );
            wFilledFrame_private:
                record

                    lineColor   :dword;
                    fillColor   :dword;

                    _linePen    :dword;
                    _lineBrush  :dword;
                    _fillBrush  :dword;

                endrecord;


        procedure create_wFilledFrame
        (
            wrName      :string;
            parent      :dword;
            x           :dword;
            y           :dword;
            width       :dword;
            height      :dword;
            lineColor   :dword;
            fillColor   :dword;
            bkgColor    :dword
        );  external;

        method get_fillColor;   @returns( "eax" );      external;
        method get_lineColor;   @returns( "eax" );      external;
```

```
            method set_fillColor( fillColor:dword );        external;
            method set_lineColor( lineColor:dword );        external;

            override method destroy;                        external;
            override method processMessage;                 external;

        endclass;
```

`lineColor` This is the RGB color of the pen used to draw the outline of the graphic object. Applications must not access this field directly but should, instead, use the accessor/mutator functions because those function maintain the private `_linePen` and `_lineBrush` fields as well.

`fillColor` This is the RGB color of the brush used to fill the interior of the graphic object. Applications must not access this field directly but should, instead, use the accessor/mutator functions because those function maintain the private `_fillBrush` field as well. Note that the `fillColor` differs from the background color (inherited from `wSurface_t`) in that the fill color paints the interior of the graphic object while the background color paints the exterior of the graphic object (within the bounding rectangle).

`_linePen`,
`_lineBrush`,
`_fillBrush` These are private fields in the class that applications must not access. These fields are automatically maintained by HOWL whenever you call one of the `wFilledFrame_t` mutator functions.

`create_wFilledFrame`
This is the constructor for the wFilledFrame_t class. Like other abstract base classes, you do not call this constructor directly, the derived classes' constructions will call this constructor for you.

`destroy` This is the destructor for the class. It frees up storage allocated for an object and frees up the system brush resources created for the object. Applications do not normally call this destructor directly; derived class destructors will call this destructor automatically.

`get_lineColor`
`get_fillColor`,
`set_fillColor`,
`set_lineColor` These are the accessor and mutator functions for the `wFilledFrame_t` data fields. You must always call these functions to access the data fields of this class because these functions also maintain the private pen and brush fields for this class.

38.3.2.8 wabsEditBox_t

The `wabsEditBox_t` class is the abstract base class used by the classes that support textual input from the user (e.g., `wEditBox_t`, `wPasswdBox_t`, and `wTextEdit_t`). Edit boxes (and text editors) are among the more feature-rich controls provided by Windows, so it's not surprising that there are many fields and functions associated with this base class.

```
    wabsEditBox_t:
        class inherits( wVisual_t );

            var
                align( 4 );
                wabsEditBox_private:
                    record
```

```
            onChange    :widgetProc;
            onErrSpace  :widgetProc;
            onHScroll   :widgetProc;
            onVScroll   :widgetProc;
            onMaxText   :widgetProc;
            onUpdate    :widgetProc;
            onSetFocus  :widgetProc;
            onKillFocus :widgetProc;
            textColor   :dword;

        endrecord;

procedure create_wabsEditBox
(
    webName     :string;
    initialTxt  :string;
    parent      :dword;
    x           :dword;
    y           :dword;
    width       :dword;
    height      :dword;
    style       :dword;
    onChange    :widgetProc
);  external;


method get_onChange;    @returns( "eax" );      external;
method get_onErrSpace;  @returns( "eax" );      external;
method get_onHScroll;   @returns( "eax" );      external;
method get_onMaxText;   @returns( "eax" );      external;
method get_onUpdate;    @returns( "eax" );      external;
method get_onSetFocus;  @returns( "eax" );      external;
method get_onKillFocus; @returns( "eax" );      external;

method set_onChange     ( onChange   :widgetProc ); external;
method set_onErrSpace   ( onErrSpace :widgetProc ); external;
method set_onHScroll    ( onHScroll  :widgetProc ); external;
method set_onMaxText    ( onMaxText  :widgetProc ); external;
method set_onUpdate     ( onUpdate   :widgetProc ); external;
method set_onSetFocus   ( onSetFocus :widgetProc ); external;
method set_onKillFocus  ( onKillFocus:widgetProc ); external;

method get_textColor;   @returns( "eax" );      external;

method set_textColor( textColor:dword );        external;

method undo;                                    external;
method cut;                                     external;
method copy;                                    external;
method paste;                                   external;
method clear;                                   external;

method get_canUndo;     @returns( "eax" );      external;
method emptyUndoBuffer;                         external;

method get_modified;    @returns( "eax" );      external;
method set_modified( modified:boolean );        external;

method get_text( txt:string );                  external;
method a_get_text; @returns( "eax" );           external;
method set_text( txt:string );                  external;
```

```
            method get_length;  @returns( "eax" );          external;

            method get_selectedText( txt:string );          external;
            method a_get_selectedText;  @returns( "eax" );  external;

            method get_selection
            (
                var startPosn   :dword;
                var endPosn     :dword
            );  external;

            method set_selection
            (
                startPosn   :dword;
                endPosn     :dword
            );  external;

            method replace_selection
            (
                replacement :string;
                canUndo     :boolean
            );  external;


            override method processMessage;                 external;

        endclass;
```

`onChange` This is a `widgetProc` pointer that HOWL will call (if it is not NULL) whenever any change is made to the text associated with the control. Note that HOWL will call this function after the update is drawn to the screen.

`onErrSpace` This is a `widgetProc` pointer that HOWL will call (if it is not NULL) if Windows cannot allocate sufficient storage to handle the current editor operator.

`onHScroll` This is a `widgetProc` pointer that HOWL will call (if it is not NULL) whenever the user clicks on the control's horizontal scroll bar. Note that HOWL will call this function before the update is drawn to the screen.

`onVScroll` This is a `widgetProc` pointer that HOWL will call (if it is not NULL) whenever the user clicks on the control's vertical scroll bar. Note that HOWL will call this function before the update is drawn to the screen.

`onMaxText` This is a `widgetProc` pointer that HOWL will call (if it is not NULL) whenever the user exceeds the maximum number of character for the edit control.

`onUpdate` This is a `widgetProc` pointer that HOWL will call (if it is not NULL) whenever any change is made to the text associated with the control. Note that HOWL will call this function before the update is drawn to the screen (this is the crucial difference between this function and the `onChange` handler).

`onSetFocus` This is a `widgetProc` pointer that HOWL will call (if it is not NULL) whenever focus shifts to the edit control.

`onKillFocus` This is a `widgetProc` pointer that HOWL will call (if it is not NULL) whenever focus shifts away from the edit control.

`textColor` This is the RGB color that Windows will use to draw the text on the editbox. This is a private data field; applications should only access this value using the associated access and mutator. The constructor initializes the text color to black.

`get_onChange,`

`set_onChange,`

get_onErrSpace,	
set_ onErrSpace,	
get_onHScroll,	
set_onHScroll,	
get_onVScroll,	
set_onVScroll,	
get_onMaxText,	
set_onMaxText,	
get_onUpdate,	
set_onUpdate,	
get_onSetFocus,	
set_onSetFocus,	
get_onKillFocus,	
set_onKillFocus	These are the accessor and mutator functions for all the data fields specific to this abstract base class. Applications should call these functions to access the data fields rather than accessing them directly.
canUndo	This function returns true in EAX if it is possible to undo the last operation to the editBox buffer (via the undo method).
emptyUndoBuffer	This method clears the undo buffer and sets the canUndo flag to false.
undo,	
cut,	
copy,	
paste,	
clear	These methods perform the standard Windows editing functions on the current selection in an edit control. Normally, you'd call these functions when the user selects an appropriate "edit" menu entry or they press one of the standard accelerator keys (e.g., "control-C" for copy).
get_text	This function retrieves the string (text) associated with an editor control and stores the text into the string argument passed as the parameter. The string passed as an argument must be large enough to hold the text or this function will raise an ex.StringOverflow exception.
a_get_text	This function retrieves the string (text) associated with an editor control and stores the text into a string allocated on the heap. It is the caller's responsibility to free the storage associated with this string when it is done using it. This function returns a pointer to the new string in the EAX register.
set_text	This function replaces the text in the edit control with the string passed as an argument.
get_length	This function returns the current number of characters in the string associated with the edit control.
get_selection	This function retrieves the starting and ending zero-based indexes into the string of the current text selection of the edit control. These indexes are returned in the two arguments passed by value. Note that the ending index will contain the offset to the character just beyond the selection in the edit control.
set_selection	This function sets the starting and ending indexes for the edit control.
replace_selection	This function replaces the selected text in the editBox with the string you supply as the argument.
get_textColor,	
set_textColor	These accessor/mutator functions get and set the text color that the widget uses.
processMessage	This is a private method in the wabsEditBox_t class. Applications should not directly call this method.

38.3.2.9 wContainer_t

The `wContainer_t` abstract base class, as its name suggests, is a special class that can contain other widgets. Containers possess a special linked list of `wBase_t` objects and the wContainer_t provides methods to manipulate this list of objects. Classes derived from `wContainer_t` include form classes, windows, radio sets, and group boxes.

```
wContainer_t:
    class inherits( wVisual_t );

        var
            align( 4 );
            wContainer_private:
                record

                    numWidgets  :uns32;
                    widgetList  :wVisual_p;
                    lastWidget  :wVisual_p;

                endrecord;

        procedure create_wContainer
        (
            wcName  :string;
            parent  :dword;
            x       :dword;
            y       :dword;
            width   :dword;
            height  :dword
        );  external;

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

        method get_numWidgets;  @returns( "eax" );      external;
        method insertWidget( theWidget:wBase_p );       external;
        method findWidget( objectID:dword );            external;
        iterator widgetOnForm( nestingLevel:uns32 );    external;
        iterator widgetsJustOnForm;                     external;

    endclass;
```

`numWidgets` This is the number of widgets contained by the `wContainer_t` object. The constructor initializes this field to zero and inserting widgets into the container increments this field by one. This is a private field; use the `get_numWidgets` method to retrieve this field's value. Applications should never store a value directly into this field.

`widgetList` This is a pointer to the first item in the list of widgets contained by the `wContainer_t` object. This is a private field; applications should never access this field.

`lastWidget` This is a pointer to the last item in the list of widgets contained by the `wContainer_t` object. This is a private field; applications should never access this field.

`destroy` This method iteratively calls the destructor for all widgets contained by the `wContainer_t` object and then it frees the storage held by the container object itself. Because `wContainer_t` is an abstract base class, applications should not call this destructor directly. Instead, they will call the destructor for some derived class which will indirectly call this method. Imporant note: because a container automatically calls the destructor for all widgets contained by the container, an application must not explicitly call the destructor for any of those widgets.

`show` This method iteratively calls all the `show` methods for each of the widgets contained by the container. Because `wContainer_t` is an abstract base class, applications should not call this method directly. Instead, they will call the `show` method for some derived class which will indirectly call this method.

`hide` This method iteratively calls all the `hide` methods for each of the widgets contained by the container. Because `wContainer_t` is an abstract base class, applications should not call this method directly. Instead, they will call the `hide` method for some derived class which will indirectly call this method.

`enable` This method iteratively calls all the `enable` methods for each of the widgets contained by the container. Because `wContainer_t` is an abstract base class, applications should not call this method directly. Instead, they will call the `enable` method for some derived class which will indirectly call this method.

`disable` This method iteratively calls all the `disable` methods for each of the widgets contained by the container. Because `wContainer_t` is an abstract base class, applications should not call this method directly. Instead, they will call the `disable` method for some derived class which will indirectly call this method.

`get_numWidgets` This method returns the value of the `numWidgets` field in the EAX register. Note that there is no corresponding "set_numWidgets" mutator function; applications cannot directly set the value of this field, `wContainer_t` objects increment the value of this field by calling the `insertWidget` method.

`insertWidget` This method inserts a widget into the `wContainer_t's` linked list. The argument is a pointer to a `wVisual_t` object (or some object type derived from `wVisual_t`). Widgets are inserted into the linked list at the end of the list (i.e., after the widget pointed at by `lastWidget`). As this is being written, there is no way to remove a widget from a `wContainer_t's` widget list. This restriction may be relaxed in a future version of HOWL.

`findWidget` This function searches for a widget in the `wContainer_t's` widget list. The single argument is the `ObjectID` value (inherited from `wBase_t`) of the object to search for. This function is mainly useful for various Windows callback functions (message handlers) that pass along a widgets object identifier without specifying the object itself.

widgetOnForm This is an HLA iterator (that you use in an HLA **foreach** loop) that iterates over all the widgets in a container's widget list. This iterator is recursive. This means that if one of the items in a widget list is a `wContainer_t` class (or a class derived from `wContainer_t`), then the iterator will drill down into that container and return its list of widgets as well. For example, the `wContainer_t destroy`, `show`, `hide`, `enable`, and `disable` methods all use this iterator to process all the widgets held by the container. On each iterator of the foreach loop, this iterator returns a pointer to the current widget in the EAX register.

`widgetsJustOnForm`

This iterator is very similar to `widgetOnForm` except that it is not recursive. On each iteration of the foreach loop it will return an entry from the current container's widget list. It will not recursively process the lists of any `wContainer_t` objects appearing in the current container.

38.3.3 Containers

There are five main (concrete) container objects in HOWL: `wForm_t` objects, window_t objects, `wGroupBox_t` objects, `wTabs_t` objects, and `wRadioSet_t` objects. We'll consider the first three of these objects in this section (plus menu objects, because it makes sense to discuss menus along with `wForm_t` objects) and `wRadioSet_t` objects in the section on buttons.

38.3.3.1 Forms and Menus

The main form for an application is a `wForm_t` object. Most applications will have a single `wForm_t` object, though a multi-window application can certainly support two or more `wForm_t` windows. A `wForm_t` menu is special (compared, say, to a `wTabPage_t` object) because it supports a menu. In a sense, a `wForm_t` object is a double container because it can contain an arbitrary list of widgets and it can contain a list of menu items.

38.3.3.1.1 wForm_t

The `wForm_t` class type is a `window_t` object with the addition of a list of menu items (which may be empty). The `wForm` statement in the HOWL declarative language defines a class that is derived from `wForm_t`. The `wForm` statement inserts the widget declarations into this new class and creates a constructor for the new class. Therefore, `wForm_t` is the basis for all forms created with the HOWL declarative language. Technically, `wForm_t` is a concrete class , not an abstract class, (meaning you can create objects of type `wForm_t`). However, in most HOWL applications your main window will actually consist of an object whose type is derived from `wForm_t` (the `wForm..endwForm` declaration creates this class for you).

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

                    menuList    :wMenuItem_p;

                endrecord;

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

        method appendMenuItem( mi:wMenuItem_p );    external;
        override method insertWidget;               external;
        override method processMessage;             external;

    endclass;
```

`menuList` This is a private data field that points at the list of menu items for the wForm_t's menu. Applications should not access this field.

`create_wForm` This is the constructor for `wForm_t` objects. If you call this procedure using the classname, e.g., "wForm_t.create_wForm(...);" then this constructor will allocate storage for the `wForm_t` object on the heap and initialze all the fields of the `wForm_t` object with reasonable values (including the parameter values you specify). If you have a statically declared wForm_t object, or a pointer to a wForm_t object you've already allocated storage for, then calling this procedure via that object will initialize the fields of that object, e.g., "somewFormObject.create_wForm(...);", without allocating new storage for the object. Although `wForm_t` is a concrete class and it's not unreasonable for an application to call `create_wForm` directly, most applications will actually work with classes derived from `wForm_t`, so it would be unreasonable for an application to call this constructor directly. If you look back at the discussion of the `wForm..endwForm` statement in the section on the HOWL declarative language, you'll notice that the `appStart` procedure calls a constructor named "myForm.create_myForm". This constructor is a good example of a constructor for a class (`myForm_t`) derived from `wForm_t`. Note that `myForm` is a statically declared object of type `myForm_t` (which is derived from `wForm_t`) in the examle given earlier in this documentation.

wwName: this is a string that HOWL stores in the `_name` field of the object (from `wBase_t`). HOWL does not copy this string, so the character data associated with this argument must exist for the duration of the program (and must not change). You should use `str.a_cpy` to create a copy of this string to pass to `create_wForm` if the original string might change during the execution of the program.

caption: This is the string that HOWL will display in the title bar of the `wForm_t's` window on the screen. Note that Windows will create an internal copy of this string, so it need not continue to exist after the call to the constructor.

exStyle: the constructor logically-ORs the value you supply to this parameter with the Windows' `w.WS_EX_CONTROLPARENT` extended window style. Normally you would supply zero for this parameter value. However, if you want your form to have some additional window extended style attributes, you can supply one (or more) of the `w.WS_EX_*` constants here. See the Windows documentation for `w.CreateWindowEx` for more details on the possible extended style constants you can use.

style: the constructor logically-ORs this value with the (`w.WS_CLIPCHILDREN | w.WS_OVERLAPPEDWINDOW`) style when creating the window. Normally you would supply zero for this parameter value. However, if you want your form to have some additional window style attributes, you can supply one (or more) of the `w.WS_*` constants here. See the Windows documentation for `w.CreateWindow` for more details on the possible window style constants you can use.

parent: this is the handle of the parent window for this form. This should always be NULL (if you are creating a child window, you'll probably be using the `window_t` type, not a `wForm_t` type).

x, y, width, height: These fields describe the position and size of the `wForm_t` window on the main screen. These will either be the pixel coordinates and sizes or the Windows' constant `w.CW_USEDEFAULT` (that tells Windows to pick good default values for these arguments).

fillColor: This is the RGB background color you want to use for the client (drawing) area of the `wForm_t` window.

visible: if true, then the constructor makes this form visible when it creates it. If this argument is false, then you must explicitly call the `show` method to make the form visible on the screen. For the main form, this argument is almost always true. If you are creating multiple windows (forms), then you might set this argument to false for all but the main form and call the show method to display the windows as needed.

`appendMenuItem` This method appends a new menu item to the `wForm_t` object's menu list. See the discussion of menu items (following shortly) for a complete discussio of those objects. In most HOWL applications, you will not directly call this method; the HOWL declarative language automatically appends all menu items you declare in the `wMenu..endwMenu`

statement to the main window's `wForm_t` object. However, if you want to manually create a `wForm_t` object, you can call this functiont to attach a menu item to the form.

`insertWidget` See `wContainer_t.insertWidget` for more details. Note that this overridden version will also set the `parentForm` field of the widget you insert (plus all contained widgets, if the argument is a container) to the value of the form.

`processMessage` This is an internal HOWL method. Applications should not call this function.

38.3.3.1.2 wMenu_t

Exactly one `wMenu_t` object is create for every `wForm_t` object that has a menu. The `wMenu_t` object corresonds to the main menu for the form. This should be the first menu item (note that `wMenu_t` is derived from `wMenuItem_t`, described next) added to the `wForm_t` object via an `appendMenuItem` method call.

```
wMenu_t:
    class inherits( wMenuItem_t );

        // Constructors/Destructors:

        procedure create_wMenu
        (
            wmName          :string;
            wmText          :string;
            parentHandle    :dword
        );  external;


        override method destroy;                    external;

    endclass;
```

`create_wMenu` This constructor creates the main menu item.

wmName: this is a string that HOWL stores in the `_name` field of the main menu object (from `wBase_t`). HOWL does not copy this string, so the character data associated with this argument must exist for the duration of the program (and must not change). You should use `str.a_cpy` to create a copy of this string to pass to `create_wMenu` if the original string might change during the execution of the program.

wmText: This is basically ignored and should be the empty string or some string like "main menu".

parentHandle: This must be the handle of the wForm_t object that contains this menu.

`destroy` This is the destructor method for the `wMenu_t` object. Applications should not call this method directly if the menu is on a form. The `wForm_t` object that holds the menu will automatically call the destructors for all the objects on its menu list (including the `wMenu_t` object).

38.3.3.1.3 wMenuItem_t

`wMenuItem_t` objects generally correspond to the actual menu items present in the main window.

```
wMenuItem_t:
    class inherits( wBase_t );

        var
            align( 4 );
            wMenuItem_private:
                record
```

```
                    nextMenu        :wMenuItem_p;
                    itemType        :dword;
                    itemString      :string;
                    itemHandler     :widgetProc;

                endrecord;


            // Constructors/Destructors:

            procedure create_wMenuItem
            (
                wmiName         :string;
                parentHandle    :dword;
                itemType        :dword;
                itemString      :string;
                itemHandler     :widgetProc
            );  external;


            override method enable;                             external;
            override method disable;                            external;

            method checked( state:boolean );                    external;

            // Accessor functions:

            method get_itemType;        @returns( "eax" );      external;
            method get_itemString;      @returns( "eax" );      external;
            method get_itemHandler;     @returns( "eax" );      external;


            method set_itemType( itemType:dword );              external;
            method set_itemString( itemString:string );         external;
            method set_itemHandler( itemHandler:widgetProc );   external;

        endclass;
```

`nextMenu` This is a private data field that the wForm_t class uses to create a linked list of menu items on the main form. Applications should not access this field.

`parentHandle` This is the handle of the main application's form

`itemType` This field specifies the type of the menu item. It must be the logical OR of oe or more of the following constants: `w.MF_STRING`, `w.MF_CHECKED`, `w.MF_DISABLED`, `w.MF_ENABLED`, `w.MF_GRAYED`, `w.MF_SEPARATOR`, and `w.MF_UNCHECKED`. See the Windows documentation for `w.AppendMenu` for more details.

`itemString` This is the string text that Windows displays for the menu item.

`itemHandler` This is a `widgetProc` procedure that HOWL will call when the user selects a menu item.

`enable`,

`disable` These two methods will enable or disable a menu item, respectively. Note that when HOWL disables a menu item, it will gray that menu item and will prevent the user from selecting it in the menu. This is equivalent to the `(w.MF_DISABLE | w.MF_GRAYED)` item type.

checked	If the current menu item has the w.MF_CHECKED flag, then calling this method will display a check mark if the argument is true, it will clear a displayed check mark if the argument is false.
get_itemType, get_itemString, get_itemHandler	These accessor functions return the values of the respective fields in the EAX register.
set_itemType, set_itemString, set_itemHandler	These mutator functions set the values of the respective fields to the value passed as an argument. Note that the set_itemString function does not make a copy of the string data, it stores the string pointer directly into the itemString data field. Therefore, your application should make a copy of the string to pass to this mutator if the string data could change.

38.3.3.2 Tabbed Forms

A tabbed form is a wForm_t object that contains exactly one wTabs_t object on it. A wTabs_t object is a container that contains a list of wTabPage_t (window) objects on it, one wTabPage_t object for each tab. present on the wTabs_t control. Whenever the user selects one of the tabs on the wTabs_t control, HOWL will display the corresponding wTabPage_t object (hiding the previously displayed wTabPage_t object). This lets an application have multiple pages on the main form that the user can select the pages as needed.

38.3.3.2.1 wTabs_t

The wTabs_t class is a special type of container class. If used correctly, wTabs_t objects only hold wTabPage_t objects (called "pages") that correspond to a window on top of the main form that hold the widgets associated with a tab on that main form. wTabs_t objects represent the current state of a tabbed form.

```
wTabPage_array :pointer to wTabPage_p;

wTabs_t:
    class inherits( wContainer_t );

        var
            align( 4 );
            wTabs_private:
                record

                    curSelection    :dword;
                    numTabs         :uns32;
                    pages           :wTabPage_array;
                    numElements     :dword;

                endrecord;

        procedure create_wTabs
        (
            wtName      :string;
            parent      :window_p;
            x           :dword;
            y           :dword;
            width       :dword;
            height      :dword
        );  external;


        method get_numTabs;                 @returns( "eax" );  external;
        method curTab;                      @returns( "eax" );  external;
```

```
    method setTab( tab:uns32 );                                 external;
    method get_page( tabIndex:dword );  @returns( "eax" );  external;
    method deleteTab( tabIndex:dword ); @returns( "eax" );  external;
    method insertTab
    (
        index   :dword;
        tabText :string;
        page    :wTabPage_p
    );  external;

    override method destroy;                                    external;
  override method processMessage;                       external;

endclass;
```

`curSelection` This field contains the zero-based tab index for the currently active tab on a `wForm_t` tabbed object. This is a private data field that applications must not access.

numTabs This private data field contains the number of tabs currently associated with the `wTabs_t` object. Applications should never directly access this data field. They can use the `get_numTabs` accessor method to query its value. Applications must never directly change the value of this field.

`pages` The `pages` data field is a pointer to an array of `wTabPage_p` pointers. This is a private data field and applications should not access or modify its contents. Applications can use the `get_page` method to read entries from the array pointed at by pages.

`numElements` This is a private data field that specifies the number of elements in the array pointed at by the pages data field. Applications must not access or modify this value.

`create_wTabs` This is the constructor for the `wTabs_t` class. If called as a class procedure (e.g., "wTabs_t.create_wTabs") then this procedure will allocate storage on the heap for a new `wTabs_t` object and return a pointer to that new object in ESI. If you call this method via an object variable (e.g., "myTab.create_wTab") then this constructor will initialize the fields of that object without allocating new storage for it (and return a pointer to the object in ESI).

wtName: this string is assigned to the `_name` field of the object. This string should not change during the execution of the program. Pass a copy of the string (using str.a_cpy) if it is possible for this string to change during program execution.

parent: this is the handle of the parent `window_t` object (e.g., `wForm_t` object) that holds this `wTabs_t` object.

x, y, width, height: These arguments specify the position and size of the tab control within the client area of the parent object. The x and y values are almost always zero, the width should be the width of the client area of the parent window, and the height should 25 or some other similar value.

`destroy` This is the destructor for the `wTabs_t` class. Generally, tabs are attached to a `wForm_t` object and that object will destroy the tabs when it is destroyed. Therefore, an application will rarely call a `wTabs_t` destructor unless it explicitly creates the `wTab_t` object and doesn't attach those tabs to some other container (e.g., `wForm_t`) object).

`get_numTabs` This method returns the current number of tabs on the tab control. In addition to returning the value of the `numTabs` data field, this method also does a sanity check to ensre that the number of tabs on the Windows control matches the `numTabs` data field value.

`curTab` This method returns the currently selected tab index (0..numtabs-1). Note that this method actually calls Windows to retrieve this value, it is not simply an accessor to the `curSelection` data field (indeed, this is a mutator to the `curSelection` field because it will update `curSelection` with the value that Windows returns). This also does a sanity check on the values and raises an exception if Windows and HOWL have different ideas about the number of tabs on the control.

setTab	This method sets the currently selected tab to the value you pass as an argument. This value must be in the range 0..numTabs-1. If the argument is outside this range, the setTab method raises an exception. This method also does a sanity check to ensure that Windows' and HOWL's tab counts are the same.
deleteTab	This function deletes a tab (specified by the zero-based tabIndex argument) from the tab bar on the form. Note that this function does not call the destructor for the tab object, nor does it destroy any of the widgets contained on the form. It simply removes the tab from the tab bar (and the corresponding entry from the array pointed at by the pages data field). This function returns the pointer to the wTabPage_t widget removed from the pages array in the EAX register. Note that this method does not remove the wTabPage_t object from the tab's widgetList (that is, the tab still contains the wTabPage_t object). Therefore, if you destroy the tab, or otherwise iterate over all the widgets held by the wTabs_t container, you will still process the deleted tab. All that deleting a tab does is visibly remove it from the tab control on the form. Note that you can insert the deleted wTabPage_t object (whose address is returned in EAX by deleteTab) by calling insertTab. It is your responsibility to save the value returned by deleteTab (or otherwise locate the deleted item) if you intend to reinsert it into the tab control later on.
insertTab	This build adds a new tab entry to the tab control and inserts a pointer to a wTabPage_t object into the array pointed at by the tab's pages data field. **index**: This is the zero-based index specifying the tab position. This value must be in the range 0..numTabs. If index is less than numTabs, then insertTab will insert the new tab in front of the tab at the specified index. If index is equal to numTabs, then insertTab will append the new tab to the end of the tab list. If index is greater than numTabs, insertTab will raise an exception. **tabText**: this is the string that Windows will draw on the tab. Windows will make a copy of this string's character data. **page**: this is the wTabPage_t object that HOWL will display when you select the new tab. Generally, a program will place several other widgets on thie wTabPage_t display surface
processMessage	This is a private method. Applications should never call this method..

38.3.3.3 wGroupBox_t

A wGroupBox_t object is a rectangular panel with a caption along the upper-left-hand corner of the rectangle (on top of the line outlining the rectangle). Generally, wGroupBox_t objects are used to visually separate and group items on a form.

```
wGroupBox_t:
    class inherits( wContainer_t );


        procedure create_wGroupBox
        (
            wgbName     :string;
            caption     :string;
            parent      :dword;
            x           :dword;
            y           :dword;
            width       :dword;
            height      :dword
        );  external;


    endclass;
```

create_wGroupBox	This is the constructor for the wGroupBox_t class.

wgbName: This is the name of the object that HOWL stores into the `_name` data field.

caption: This is the string that Windows displays in the upper-left-hand corner of the group box. Windows makes a copy of this string for its internal use.

parent: This is the handle of the window (usually the main form or a `wView_t` object on a tabbed form) that contains the group box.

x, y, width, height: These are the (parent-form-relative) coordinates and size for the group box.

38.3.4 Graphic Objects

Graphic objects in HOWL are static images that HOWL draws on a form. Examples include rectangles and ellipses. All graphic objects are derived from `wSurface_t`, which is derived from `wClickable_t`, so graphic objects can respond to single clicks. Note that although the `wClickable_t` type also handles double clicks, graphic objects don't send double-click notifications, so if you try to install a double-click handler for one of these objects (which is legal to do), it won't have any effect. Double-clicks will simply be treated as two single clicks. Because attaching an "onClick" handler to a graphic object is not the common case, you will have to explicitly call the `set_onClick` method to initialize an `onClick` handler. The HOWL declarative language doesn't provide an option to do this for you.

38.3.4.1 wBitmap_t

The `wBitmap_t` class lets you create objects that display a bit mapped image on a form.

```
wBitmap_t:
    class inherits( wSurface_t );

        var
            align( 4 );
            wBitmap_private:
                record

                    stretch         :boolean;
                    align( 4 );

                    imageName       :string;
                    imageHandle     :dword;
                    sourceX         :dword;
                    sourceY         :dword;
                    sourceW         :dword;
                    sourceH         :dword;
                    destW           :dword;
                    destH           :dword;

                endrecord;


        procedure create_wBitmap
        (
            wiName      :string;
            imageName   :string;
            parent      :dword;
            x           :dword;
            y           :dword;
            width       :dword;
            height      :dword;
            bkgColor    :dword
        );  external;

        method get_imageName;       @returns( "eax" );      external;
```

```
        method get_sourceX;         @returns( "eax" );      external;
        method get_sourceY;         @returns( "eax" );      external;
        method get_sourceW;         @returns( "eax" );      external;
        method get_sourceH;         @returns( "eax" );      external;
        method get_destW;           @returns( "eax" );      external;
        method get_destH;           @returns( "eax" );      external;
        method get_stretch;         @returns( "al" );       external;

        method load_bitmap( imageName:string );             external;
        override method destroy;                            external;
        override method processMessage;                     external;

        method normalBitmap;        @returns( "eax" );      external;
        method stretchBitmap
        (
            sourceX         :dword;
            sourceY         :dword;
            sourceW         :dword;
            sourceH         :dword;
            destW           :dword;
            destH           :dword
        );  external;


    endclass;
```

`imageName` This is a string that specifies the name of the bitmap resource within the executable file. Important this is not the name of a ".bmp" file on the disk. You must compile ".bmp" files into your executable using a "resouce compiler". The name you attach to the resource is the name that this string will contain. This is a private field. Applications should not access it directly. Note that if the value of this field is less than $1_0000, then it specifies a standard Windows bitmap resource rather than an actual resource name. See the discussion of the load_bitmap method for more details.

`imageHandle`,
`sourceX`,
`sourceY`,
`sourceW`,
`sourceH`,
`destW`,
`destH` These are private data fields that applications must not access or modify.

`stretch` This private field determines whether a bitmap is stretched or displayed normal. This field is set to true by the `stretchBitmap` method and set to false by the `normalBitmap` method.

`create_wBitmap` This is the constructor for the wBitmap_t class. If called as a class procedure (e.g., "wBitMap_t.create_wBitmap") this procedure will allocate storage on the heap for the object and return a pointer to the new (initialized) object in ESI. If you call this constructor specifying an existing object, then it will simply initialize that object in-place.

wiName: this is the string that the HOWL code stores into the `_name` field. This string's value should not change over the lifetime of the bitmap object.

imageName: This is either a string containing a bitmap resource name within the executable file, or a standard Windows bitmap resource value. Legal bitmap resource constants are:

w.OBM_BTNCORNERS, OBM_BTSIZE, w.OBM_CHECK, w.OBM_CHECKBOXES, w.OBM_CLOSE, w.OBM_REDUCE, w.OBM_COMBO, w.OBM_REDUCED, w.OBM_DNARROW, w.OBM_RESTORE, w.OBM_DNARROWD,

w.OBM_RESTORED, w.OBM_DNARROWI, w.OBM_RGARROW, w.OBM_LFARROW, w.OBM_RGARROWD, w.OBM_LFARROWD, w.OBM_RGARROWI, w.OBM_LFARROWI, w.OBM_SIZE, w.OBM_MNARROW, w.OBM_UPARROW, w.OBM_UPARROWD, w.OBM_UPARROWI, w.OBM_ZOOM, w.OBM_ZOOMD.

See the Windows documentation for more details on these constants.

parent: this is the handle of the form, `wView_t`, or other drawing surface that contains the wBitmap_t object (and on whose surface the bitmap will be drawn).

x, y, width, height: these arguments specify the bounding box on the parent's form where the bitmap will be drawn. If this bounding rectangle is larger than the bitmap image (in any dimension), then HOWL will fill the unaccounted-for area wth the background color. If this bounding rectangle is smaller than the image (in any dimension), then HOWL will clip the bitmap when drawing it.

bkgColor: this is the RGB background color that HOWL uses to fill in the bounding rectangle if the bitmap is smaller than the bounding rectangle.

`load_bitmap` This method loads the bitmap object with an image resource in the executable file. The argument is a string specifying the resource name or one of the standard Windows bitmap resource constants (see the discussion in `create_wBitmap`). Note that this is not a ".bmp" filename. You must compile bitmaps into the executable file using a resource compiler and specify the resource name as the parameter to `load_bitmap`.

`destroy` This is the class destructor. This method releases all resources and memory in use by the `wBitmap_t` object. Usually, applications will not call this method directly. Instead, HOWL will automatically call this destructor when destroying the main application's `wForm_t` form or a `wView_t` object that contains the bitmap.

`processMessage` This is a private method that applications should never call.

`normalBitmap` This method sets the `stretch` field to false to display the bitmap in a normal form.

stretchBitmap This method sets the stretch field to true and copies the parameters to the corresponding private data fields.

sourceX: The stretched bitmap will be copied from the original bit map starting at this zero-based x-coordinate.

sourceY: The stretched bitmap will be copied from the original bit map starting at this zero-based y-coordinate.

sourceW: This many bits along the X axis will be copied to the stretched bitmap.

sourceH: This many bits along the Y axis will be copied to the stretched bitmap.

destW: The bit mapped will be stretched (or shrunk) so that the `sourceW` bits will be displayed using `destW` bits.

destH: The bit mapped will be stretched (or shrunk) so that the `sourceH` bits will be displayed using `destH` bits.

38.3.4.2 wEllipse_t

Ellipse graphic objects (of which circles are special cases) allow you to place ellipses anywhere on a form or `wView_t` object.

```
wEllipse_t:
    class inherits( wFilledFrame_t );

        procedure create_wEllipse
        (
            wrName      :string;
            parent      :dword;
            x           :dword;
```

```
            y           :dword;
            width       :dword;
            height      :dword;
            lineColor   :dword;
            fillColor   :dword;
            bkgColor    :dword
        );  external;

        override method processMessage;                 external;

    endclass;
```

`create_wEllipse` This is the consructor for the `wEllise_t` class. If you call this as a class procedure (e.g., "wEllipse_t.create_wEllipse") then this procedure will allocate storage for a new `wEllipse_t` object on the heap and return a pointer to that object in ESI. If you make a standard object call to this constructor, then `create_wEllipse` will initialize that object in-place.

wrName: HOWL assigns this string to the _name data field of the object. This string's value should be constant over the execution lifetime of the newly initialized object.

parent: this is the handle of the `wView_t` or `wForm_t` object on which the ellipse will be drawn.

x, y, width, height: These arguments form a bounding box in which the ellipse will be drawn. If `width` and `height` are the same value (meaning the bounding rectangle is a square), then the ellipse will form a circle.

lineColor: this is the RGB color value for the pen that HOWL will use to draw the outline of the ellipse.

fillColor: this is the RGB color value for the brush that HOWL will use to paint the interior of the ellipse.

bkgColor: this is the RGB color value for the brush that HOWL will use to paint the exterior of the ellipse.

`processMessage` This is a private method that applications must not call.

38.3.4.3 wPie_t

The `wPie_t` graphic object draws a slice of a pie graph on a window. Note that `wPie_t` is only capable of drawing a single wedge of a pie graph. **Note**: the procedures and methods in the `wPie_t` class make use of the FPU on the CPU. You must ensure that the FPU is initialized (i.e., you're not in MMX mode) before using these functions.

```
wPie_t:
    class inherits( wFilledFrame_t );

        var
            align( 8 );
            wPie_private:
                record

                    startAngle  :real64;
                    endAngle    :real64;

                endrecord;

        procedure create_wPie
        (
            wrName      :string;
```

```
                parent      :dword;
                x           :dword;
                y           :dword;
                width       :dword;
                height      :dword;
                startAngle  :real64;
                endAngle    :real64;
                lineColor   :dword;
                fillColor   :dword;
                bkgColor    :dword
            );  external;


            override method processMessage;                 external;

            method get_startAngle;  @returns( "st0" );      external;
            method get_endAngle;    @returns( "st0" );      external;

            method set_startAngle( startAngle:real64 );     external;
            method set_endAngle( endAngle:real64 );         external;

        endclass;
```

`startAngle` This is the starting angle (measure counter-clockwise from the vertical line) from which `wPie_t` objects between drawing the wedge. Applications should not access or modify this field directly; they should use the supplied accessor and mutator functions for this purpose. Do not assume the value of this field is degrees or radians.

endAngle This is the ending angle (measured counter-clockwise from the vertical line) to which wPie_t object draw the wedge (from the startAngle to the endAngle in the counter-clockwise direction). Applications should not access or modify this field directly; they should use the supplied accessor and mutator functions for this purpose. Do not assume the value of this field is degrees or radians.

`create_wPie` This the is the constructor for the `wPie_t` class. If you call this as a class procedure (e.g., "wPie_t.create_wPie") then this procedure will allocate storage for a new `wPie_t` object on the heap and return a pointer to that object in ESI. If you make a standard object call to this constructor, then `create_wPie` will initialize that object in-place.

wrName: HOWL assigns this string to the _name data field of the object. This string's value should be constant over the execution lifetime of the newly initialized object.

parent: this is the handle of the `wView_t` or `wForm_t` object on which the wedge will be drawn.

x, y, width, height: These arguments form a bounding box in which the pie wedge will be drawn.

startAngle: starting angle for the wedge (see the discussion above). This angle is specified in degrees (not radians).

endAngle: ending angle for the wedge (see the discussion above). This angle is specified in degrees, not radians.

lineColor: this is the RGB color value for the pen that HOWL will use to draw the outline of the wedge.

fillColor: this is the RGB color value for the brush that HOWL will use to paint the interior of the wedge.

bkgColor: this is the RGB color value for the brush that HOWL will use to paint the exterior of the wedge.

`processMessage` This is a private method that applications must not call.

`get_startAngle`,

`get_endAngle` These are accessor functions for the `startAngle` and `endAngle` data fields. Note that because these values are real, these functions return their results on the top of the FPU stack. These functions return the angles in degrees.

`set_startAngle`, `set_endAngle` These are the mutator functions for the `startAngle` and `endAngle` data fields. These functions expect the angle in degrees.

38.3.4.4 wPolygon_t

A polygon is a closed geometric object created by drawing a set of lines between the points in a list (and from the last point to the first point to close the object).

Note: Windows automatically resizes most geographic objects you draw (e.g., rectangles and ellipses). It does not, however, resize a polygon if you change its bounding box. The HOWL polygon class, fortunately, contains extra code to resize a polygon if you change the width or height of the bounding box. Therefore, when using HOWL, your programs can treat polygons just like other geometric objects with respect to the resize method.

```
ptArray :pointer to w.POINT;  // w.POINT:[x:dword, y:dword]

wPolygon_t:
    class inherits( wFilledFrame_t );

        var
            align( 4 );
            wPolygon_private:
                record

                    points          :ptArray;
                    scaledPoints    :ptArray;
                    nPoints         :uns32;
                    origW           :dword;
                    origH           :dword;

                endrecord;

        procedure create_wPolygon
        (
            wrName      :string;
            parent      :dword;
            x           :dword;
            y           :dword;
            width       :dword;
            height      :dword;
            lineColor   :dword;
            fillColor   :dword;
            bkgColor    :dword
        );  external;


        override method destroy;                        external;
        override method processMessage;                 external;

        override method set_width;                      external;
        override method set_height;                     external;
        override method resize;                         external;

        method set_points
        (
            nPoints :dword;
```

```
                points  :ptArray
            );  external;

            method get_points;      @returns( "eax" );      external;
            method get_nPoints;     @returns( "eax" );      external;


        endclass;
```

`points` This is the address of an array of w.POINT objects in memory (each element is 8 bits, a four-byte x-coordinate followed by a four byte y-coordinate value). This is a private data field that applications should not access or modify directly.

`scaledPoints` This is the address of an array of w.POINT objects in memory (each element is 8 bits, a four-byte x-coordinate followed by a four byte y-coordinate value). The Polygon class uses this array (rather than points) to draw the polygon if the polyon has been resized since it was created. This is a private data field that applications should not access or modify directly.

`nPoints` This is the number of points in the array pointed at by the `points` field. If this field is zero, then the `points` data field may contain an arbitary value. This is a private data field that applications should not access or modify directly.

`origW` This field holds the original (created) width of the current polygon. The polygon class uses this value to determine if it has to scale the polygon along the x-axis because the polygon has been resized. This is a private data field that applications should not access or modify directly.

`origH` This field holds the original (created) height of the current polygon. The polygon class uses this value to determine if it has to scale the polygon along the y-axis because the polygon has been resized. This is a private data field that applications should not access or modify directly.

`create_wPolygon` This the is the constructor for the `wPolygon_t` class. If you call this as a class procedure (e.g., "wPolygon_t.create_wPolygon") then this procedure will allocate storage for a new `wPolygon_t` object on the heap and return a pointer to that object in ESI. If you make a standard object call to this constructor, then `create_wPolygon` will initialize that object in-place. Note that this constructor initializes `nPoints` to zero (and `points` to NULL). So immediately upon creation, the polygon has no vertexes and it will not draw anything on the form until you provide a list of points.

wrName: HOWL assigns this string to the _name data field of the object. This string's value should be constant over the execution lifetime of the newly initialized object.

parent: this is the handle of the `wView_t` or `wForm_t` object on which the polygon will be drawn.

x, y, width, height: These arguments form a bounding box in which the polygon will be drawn.

lineColor: this is the RGB color value for the pen that HOWL will use to draw the outline of the wedge.

fillColor: this is the RGB color value for the brush that HOWL will use to paint the interior of the wedge.

bkgColor: this is the RGB color value for the brush that HOWL will use to paint the exterior of the wedge.

`processMessage` This is a private method that applications must not call.

`destroy` This is the destructor for the `wPolygon_t` class. Normally, applications do not call this destructor directly; instead, a container will call this destructor automatically when the container is destroyed. However, if you've created an independent (of any container) polygon object, you should call this destructor to free the resources it uses when you are done with the polygon.

`set_width,`

`set_height`,

`resize` This are overridden methods from the `wFilledFrame_t` class. They handle scaling the polygon when you change the size of the polygon's bounding box. See the descriptions in `wFillFrame_t` for more details. Note that the these functions will usually make a copy of the points data pointed at by the `points` field and then set `scaledPoints` to point at this new data (note, however, that if you reset the size back to the original size, then these functions will deallocate the storage pointed at by the `scaledPoints` field).

`set_points` This is the mutator for the `points` and `nPonts` data fields.

nPoints: This argument specifies the number of points in the points array passed as the second argument.

points: this is a pointer to an array of `nPoints w.POINT` elements. The `set_points` method will make a copy of this data into internally allocated storage (on the heap) and store a pointer to the new data in the points field (this call also frees any storage previously in use by the `points` and `scaledPoints` fields).

`get_nPoints` This accessor returns the number of points in the polygon (the value of the `nPoints` field).

`get_points` The `get_points` accessor function returns the value of the `points` or `scaledPoints` field. It returns a pointer to the `points` field if the current bounding box `width` and `height` of the polygon haven't changed since the last `set_points` call. This method returns `scaledPoints` if the polygon has been resized.

38.3.4.5 wRectangle_t

The `wRectangle_t` graphic object displays a (clickable) rectangle on a window or form.

```
wRectangle_t:
    class inherits( wFilledFrame_t );

        procedure create_wRectangle
        (
            wrName      :string;
            parent      :dword;
            x           :dword;
            y           :dword;
            width       :dword;
            height      :dword;
            lineColor   :dword;
            fillColor   :dword
        );  external;


        override method processMessage;                 external;

    endclass;
```

`create_wRectangle`

This is the consructor for the `wRectangle_t` class. If you call this as a class procedure (e.g., "wRectangle_t.create_wRectangle") then this procedure will allocate storage for a new `wRectangle_t` object on the heap and return a pointer to that object in ESI. If you make a standard object call to this constructor, then `wRectangle_t` will initialize that object in-place.

wrName: HOWL assigns this string to the `_name` data field of the object. This string's value should be constant over the execution lifetime of the newly initialized object.

parent: this is the handle of the `window` object on which the ellipse will be drawn.

x, y, width, height: These arguments form a bounding box in which the rectangle will be drawn. If `width` and `height` are the same value, then the rectangle will form a square.

lineColor: this is the RGB color value for the pen that HOWL will use to draw the outline of the rectangle.

fillColor: this is the RGB color value for the brush that HOWL will use to paint the interior of the rectangle.

`processMessage` This is a private method that applications must not call.

38.3.4.6 wRoundRect_t

`wRoundRect_t` objects are graphic objects that are rectangles with rounded corners.

```
wRoundRect_t:
    class inherits( wFilledFrame_t );

        var
            align( 4 );
            wRoundRect_private:
                record

                    cornerWidth     :dword;
                    cornerHeight    :dword;

                endrecord;

        procedure create_wRoundRect
        (
            wrName          :string;
            parent          :dword;
            x               :dword;
            y               :dword;
            width           :dword;
            height          :dword;
            cornerWidth     :dword;
            cornerHeight    :dword;
            lineColor       :dword;
            fillColor       :dword;
            bkgColor        :dword
        );  external;

        method get_cornerWidth;     @returns( "eax" );  external;
        method get_cornerHeight;    @returns( "eax" );  external;

        method set_cornerWidth( cornerWidth:dword );    external;
        method set_cornerHeight( cornerHeight:dword );  external;

        override method processMessage;                 external;

    endclass;
```

cornerWidth This data field controls the width of the ellipse that Windows draws on each corner of the rounded rectangle. Applications should not access this field directly, they should use the appropriate accessor and mutator functions to access or set the value of this data field. This value should be less than 1/2 the height of the round rectangle object.

cornerHeight This data field controls the height of the ellipse that Windows draws on each corner of the rounded rectangle. Applications should not access this field directly, they should use the

appropriate accessor and mutator functions to access or set the value of this data field. This value should be less than 1/2 the height of the round rectangle object.

`create_wRoundRect`

This is the consructor for the `wRoundRect_t` class. If you call this as a class procedure (e.g., "wRoundRect_t.create_wRoundRect") then this procedure will allocate storage for a new `wRoundRect_t` object on the heap and return a pointer to that object in ESI. If you make a standard object call to this constructor, then `create_wRoundRect` will initialize that object in-place.

wrName: HOWL assigns this string to the _name data field of the object. This string's value should be constant over the execution lifetime of the newly initialized object.

parent: this is the handle of the `wView_t` or `wForm_t` object on which the round rectangle will be drawn.

x, y, width, height: These arguments form a bounding box in which the round rectangle will be drawn.

lineColor: this is the RGB color value for the pen that HOWL will use to draw the outline of the round rectangle.

fillColor: this is the RGB color value for the brush that HOWL will use to paint the interior of the round rectangle.

bkgColor: this is the RGB color value for the brush that HOWL will use to paint the exterior of the round rectangle (the area just outside the rounded corners).

`get_cornerWidth`, `get_cornerHeight` These are the accessor functions for the `cornerWidth` and `cornerHeight` data fields. Applications should call these methods rather than accessing the data fields directly.

`set_cornerWidth`, `set_cornerHeight` These are the mutator functions for the `cornerWidth` and `cornerHeight` data fields. Applications should call these methods rather than writing directly to the data fields.

`processMessage` This is a private method that applications must not call.

38.3.5 Buttons

The HOWL button widgets come in two basic varieties: checkable (check boxes and radio buttons) and non-checkable (push buttons). All buttons are derived from the `wClickable_t` class. The constructors for these buttons let you initialize the `onClick widgetProc` associated with all buttons; you can also call the `set_onDblClick` method to make a button double-clickable.

38.3.6 wCheckBox_t

wCheckBox_t objects hare a binary state (checked or unchecked). Whenever the user clicks on a checkbox, the widget toggles its state. Note that `wCheckBox_t` objects inherit the fields of the `wCheckable_t` class. You can call the `get_check` and `set_check` methods of that class to get the current `wCheckBox_t` object state or to set it.

```
wCheckBox_t:
    class inherits( wCheckable_t );

        procedure create_wCheckBox
        (
            wcbName     :string;
            caption     :string;
            parent      :dword;
            x           :dword;
            y           :dword;
            width       :dword;
            height      :dword;
```

```
                onClick     :widgetProc
            );  external;


        endclass;
```

`create_wCheckBox`

This is the consructor for the `wCheckBox_t` class. If you call this as a class procedure (e.g., "wCheckBox_t.create_wCheckBox") then this procedure will allocate storage for a new `wCheckBox_t` object on the heap and return a pointer to that object in ESI. If you make a standard object call to this constructor, then `create_wCheckBox` will initialize that object in-place.

wcbName: HOWL assigns this string to the _name data field of the object. This string's value should be constant over the execution lifetime of the newly initialized object.

caption: this is the caption text that will be drawn immediately to the right of the check box. Windows makes an internal copy of this string, so the value need only exist for as long as the constructor call is in progress.

parent: this is the handle of the `wView_t` or `wForm_t` object on which the check box will be drawn.

x, y, width, height: These arguments form a bounding box in which the check box and caption will be drawn.

onClick: this is the name of a `widgetProc` that HOWL will call whenever you click on the checkbox widget. If this field contains NULL, HOWL will not call any `widgetProc` procedure.

38.3.7 wCheckBox3_t

`wCheckBox3_t` checkboxes are similar to standard checkboxes except they have three states: checked, unchecked, and grayed. The `get_state` method (inherited from `wCheckable_t`) will return 0 (unchecked), 1 (checked), or 2 (grayed).

```
    wCheckBox3_t:
        class inherits( wCheckable_t );

            procedure create_wCheckBox3
            (
                wcb3Name    :string;
                caption     :string;
                parent      :dword;
                x           :dword;
                y           :dword;
                width       :dword;
                height      :dword;
                onClick     :widgetProc
            );  external;


        endclass;
```

`create_wCheckBox3`

This is the consructor for the `wCheckBox3_t` class. If you call this as a class procedure (e.g., "wCheckBox3_t.create_wCheckBox3") then this procedure will allocate storage for a new `wCheckBox3_t` object on the heap and return a pointer to that object in ESI. If you make a standard object call to this constructor, then `create_wCheckBox3` will initialize that object in-place.

wcbName: HOWL assigns this string to the _name data field of the object. This string's value should be constant over the execution lifetime of the newly initialized object.

caption: this is the caption text that will be drawn immediately to the right of the check box. Windows makes an internal copy of this string, so the value need only exist for as long as the constructor call is in progress.

parent: this is the handle of the `wView_t` or `wForm_t` object on which the check box will be drawn.

x, y, width, height: These arguments form a bounding box in which the check box and caption will be drawn.

onClick: this is the name of a `widgetProc` that HOWL will call whenever you click on the checkbox widget. If this field contains NULL, HOWL will not call any `widgetProc` procedure.

38.3.8 wCheckBox3LT_t

`wCheckBox3LT_t` checkboxes are similar to `wCheckBox3_t` checkboxes except they draw the caption text to the left of the checkbox rather than to the right of it. The `get_state` method (inherited from `wCheckable_t`) will return 0 (unchecked), 1 (checked), or 2 (grayed).

```
wCheckBox3LT_t:
    class inherits( wCheckable_t );

        procedure create_wCheckBox3LT
        (
            wcb3ltName  :string;
            caption     :string;
            parent      :dword;
            x           :dword;
            y           :dword;
            width       :dword;
            height      :dword;
            onClick     :widgetProc
        );  external;


    endclass;
```

`create_wCheckBox3LT`

This is the consructor for the `wCheckBox3LT_t` class. If you call this as a class procedure (e.g., "wCheckBox3LT_t.create_wCheckBox3LT") then this procedure will allocate storage for a new `wCheckBox3LT_t` object on the heap and return a pointer to that object in ESI. If you make a standard object call to this constructor, then `create_wCheckBox3LT` will initialize that object in-place.

wcbName: HOWL assigns this string to the _name data field of the object. This string's value should be constant over the execution lifetime of the newly initialized object.

caption: this is the caption text that will be drawn immediately to the left of the check box. Windows makes an internal copy of this string, so the value need only exist for as long as the constructor call is in progress.

parent: this is the handle of the `wView_t` or `wForm_t` object on which the check box will be drawn.

x, y, width, height: These arguments form a bounding box in which the check box and caption will be drawn.

onClick: this is the name of a `widgetProc` that HOWL will call whenever you click on the checkbox widget. If this field contains NULL, HOWL will not call any `widgetProc` procedure.

38.3.9 wCheckBoxLT_t

`wCheckBoxLT_t` checkboxes are similar to `wCheckBox_t` checkboxes except they draw the caption text to the left of the checkbox rather than to the right of it.

```
wCheckBoxLT_t:
    class inherits( wCheckable_t );

        procedure create_wCheckBoxLT
        (
            wcb3ltName  :string;
            caption     :string;
            parent      :dword;
            x           :dword;
            y           :dword;
            width       :dword;
            height      :dword;
            onClick     :widgetProc
        );  external;


    endclass;
```

`create_wCheckBoxLT`

This is the consructor for the `wCheckBoxLT_t` class. If you call this as a class procedure (e.g., "wCheckBoxLT_t.create_wCheckBoxLT") then this procedure will allocate storage for a new `wCheckBoxLT_t` object on the heap and return a pointer to that object in ESI. If you make a standard object call to this constructor, then `create_wCheckBoxLT` will initialize that object in-place.

wcbName: HOWL assigns this string to the _name data field of the object. This string's value should be constant over the execution lifetime of the newly initialized object.

caption: this is the caption text that will be drawn immediately to the left of the check box. Windows makes an internal copy of this string, so the value need only exist for as long as the constructor call is in progress.

parent: this is the handle of the `wView_t` or `wForm_t` object on which the check box will be drawn.

x, y, width, height: These arguments form a bounding box in which the check box and caption will be drawn.

onClick: this is the name of a `widgetProc` that HOWL will call whenever you click on the checkbox widget. If this field contains NULL, HOWL will not call any `widgetProc` procedure.

38.3.10 wPushButton_t

wPushButton_t objects are standard Windows push button widgets. They almost always invoke some sort of "onClick" widgetProc procedure when the button is pressed.

```
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;
```

`create_wPushButton`

This is the consructor for the `wPushButton_t` class. If you call this as a class procedure (e.g., "wPushButton_t.create_wPushButton") then this procedure will allocate storage for a new `wPushButton_t` object on the heap and return a pointer to that object in ESI. If you make a standard object call to this constructor, then `create_wPushButton` will initialize that object in-place.

wcbName: HOWL assigns this string to the _name data field of the object. This string's value should be constant over the execution lifetime of the newly initialized object.

caption: this is the caption text that will be drawn on the push button. Windows makes an internal copy of this string, so the value need only exist for as long as the constructor call is in progress.

parent: this is the handle of the `wView_t` or `wForm_t` object on which the push button will be drawn.

x, y, width, height: These arguments form a bounding box in which the push button and caption will be drawn.

onClick: this is the name of a `widgetProc` that HOWL will call whenever you click on the push button widget. If this field contains NULL, HOWL will not call any `widgetProc` procedure.

38.3.11 wRadioButton_t

`wRadioButton_t` objects are stand-alone radio buttons on a form. You'll rarely use these objects because radio buttons are generally employed in sets (using a `wRadioSet_t` container and `wRadioSetButton_t` objects). A stand-alone radio button is essentially a check box with a circle and a dot rather than a square and an "x". The main purpose for wRadioButton_t objects (and wRadioButtonLT_t objects) is for programmers who want to manually control the operation of the radio buttons.

```
    wRadioButton_t:
        class inherits( wCheckable_t );

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


        endclass;
```

`create_wRadioButton`

This is the consructor for the wRadioButton_t class. If you call this as a class procedure (e.g., "wRadioButton_t.create_wRadioButton") then this procedure will allocate storage for a new wRadioButton_t object on the heap and return a pointer to that object in ESI. If you make a standard object call to this constructor, then create_wRadioButton will initialize that object in-place.

wrbName: HOWL assigns this string to the _name data field of the object. This string's value should be constant over the execution lifetime of the newly initialized object.

caption: this is the caption text that will be drawn to the right of the radio button. Windows makes an internal copy of this string, so the value need only exist for as long as the constructor call is in progress.

parent: this is the handle of the wView_t or wForm_t object on which the radio button will be drawn.

x, y, width, height: These arguments form a bounding box in which the radio button and caption will be drawn.

onClick: this is the name of a widgetProc that HOWL will call whenever you click on the radio button widget. If this field contains NULL, HOWL will not call any widgetProc procedure. Generally, if you're using wRadioButton_t objects in your application, it is the responsibility of the onClick procedure to properly update the other radio buttons associated with the one the user has just clicked on.

38.3.12 wRadioButtonLT_t

wRadioButton_LT objects are just like wRadioButton_t objects except the text appears to the left of the radio button rather than to the right.

```
wRadioButtonLT_t:
    class inherits( wCheckable_t );

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


    endclass;
```

create_wRadioButtonLT

This is the consructor for the wRadioButton_t class. If you call this as a class procedure (e.g., "wRadioButtonLT_t.create_wRadioButtonLT") then this procedure will allocate storage for a new wRadioButtonLT_t object on the heap and return a pointer to that object in ESI. If you make a standard object call to this constructor, then create_wRadioButtonLT will initialize that object in-place.

wrbltName: HOWL assigns this string to the _name data field of the object. This string's value should be constant over the execution lifetime of the newly initialized object.

caption: this is the caption text that will be drawn to the left of the radio button. Windows makes an internal copy of this string, so the value need only exist for as long as the constructor call is in progress.

parent: this is the handle of the wView_t or wForm_t object on which the radio button will be drawn.

x, y, width, height: These arguments form a bounding box in which the radio button and caption will be drawn.

onClick: this is the name of a widgetProc that HOWL will call whenever you click on the radio button widget. If this field contains NULL, HOWL will not call any widgetProc procedure. Generally, if you're using wRadioButtonLT_t objects in your application, it is the responsibility of the onClick procedure to properly update the other radio buttons associated with the one the user has just clicked on. Note that the wRadioSet..endwRadioSet statement in the HOWL declarative language will report an error if you attempt to add some non-radio-set-button widget to the wRadioSet_t object you're creating.

38.3.13 wRadioSet_t

A wRadioSet_t object is a container that holds (only) wRadioSetButton_t and wRadioSetButtonLT_t objects. The wRadioSet_t object automatically maintains all the buttons it contains, ensuring that (at most) one button is checked at a time. Note that an application must only insert groups of wRadioSetButton_t and wRadioSetButtonLT_t objects into the widget list of a wRadioSet_t object. If an application (manually) inserts other objects into a wRadioSet_t widget list, the radio buttons may not behave properly. Visually, a wRadioSet_t object is identical to a wGroupBox_t object. That is, it is a rectangular panel with a caption in the upper-left-hand corner of the rectangle.

```
wRadioSet_t:
    class inherits( wContainer_t );

        var
            align( 4 );
            wRadioSet_private:
                record

                    // Windows handle for the group box window

                    groupBoxHndl    :dword;


                endrecord;


        procedure create_wRadioSet
        (
            wrsName     :string;
            caption     :string;
            parent      :dword;
            x           :dword;
            y           :dword;
            width       :dword;
            height      :dword;
            bkgColor    :dword
        );  external;

        override method processMessage;                     external;
        override method destroy;                            external;
        override method set_width;                          external;
        override method set_height;                         external;
        override method resize;                             external;
```

```
        endclass;
```

`groupBoxHndl` This is the handle for the actual group box (a separate surface for the background is use to fill in the area behind the group box). This is a private field. Applications should not access it.

`create_wRadioSet` This is the constructor for the wRadioSet_t class. If you call this as a class procedure (e.g., "wRadioSet_t.create_wRadioSet") then this procedure will allocate storage for a new `wRadioSet_t` object on the heap and return a pointer to that object in ESI. If you make a standard object call to this constructor, then `create_wRadioSet` will initialize that object in-place.

wrsName: HOWL assigns this string to the _name data field of the object. This string's value should be constant over the execution lifetime of the newly initialized object.

caption: this is the caption text that will be drawn in the upper-left-hand corner of the `wRadioSet_t's` panel rectangle. Windows makes an internal copy of this string, so the value need only exist for as long as the constructor call is in progress.

parent: this is the handle of the `wView_t` or `wForm_t` object on which the radio set group box will be drawn.

x, y, width, height: These arguments form a bounding box in which the radio button and caption will be drawn.

`processMessage` This is a private method. Applications must not call it.

`destroy` This is the class constructor. Usually, a container object will call this destructor automatically for you; applications don't normally call this destructor unless they create a `wRadioSet_t` object and don't insert it into some container's widget list.

`set_width`,

`set_height`,

`resize` These fields are overriden from the `wVisual_t` class. See the description there for more details.

You will want to call the `insertWidget` method (inherited from `wContainer_t`) in order to add `wRadioSetButton_t` or `wRadioSetButtonLT_t` objects to a `wRadioSet_t` object.

38.3.13.1 wRadioSetButton_t

A `wRadioSetButton_t` is a standard radio set button that appears within a `wRadioSet_t` group box. `wRadioSetButton_t` objects are identical to `wRadioButton_t` objects except that they support automatic radio button control on a `wRadioSet_t` group box. You can actually specify `wRadioSetButton_t` objects outside of a `wRadioSet_t` group box; however, HOWL will only maintain automatic radio button operation on those buttons you declare (in the HOWL declarative language) in a sequence without any other intervening widget types (except `wRadioSetButtonLT_t` objects, which can be intermixed with `wRadioSetButton_t` objects).

```
wRadioSetButton_t:
    class inherits( wCheckable_t );

        procedure create_wRadioSetButton
        (
            wrbName     :string;
            caption     :string;
            style       :dword;
            parent      :dword;
            x           :dword;
            y           :dword;
            width       :dword;
            height      :dword;
            onClick     :widgetProc
```

```
            );  external;


        endclass;
```

`create_wRadioSetButton`

This is the consructor for the `wRadioSetButton_t` class. If you call this as a class procedure (e.g., "wRadioSetButton_t.create_wRadioSetButton") then this procedure will allocate storage for a new `wRadioSetButton_t` object on the heap and return a pointer to that object in ESI. If you make a standard object call to this constructor, then `create_wRadioSetButton` will initialize that object in-place.

wrbName: HOWL assigns this string to the _name data field of the object. This string's value should be constant over the execution lifetime of the newly initialized object.

caption: this is the caption text that will be drawn to the right of the radio button. Windows makes an internal copy of this string, so the value need only exist for as long as the constructor call is in progress.

parent: this is the handle of the `wView_t` or `wForm_t` object on which the radio button will be drawn.

x, y, width, height: These arguments form a bounding box in which the radio button and caption will be drawn.

onClick: this is the name of a `widgetProc` that HOWL will call whenever you click on the radio button widget. If this field contains NULL, HOWL will not call any `widgetProc` procedure. Generally, if you're using `wRadioButton_t` objects in your application, it is the responsibility of the `onClick` procedure to properly update the other radio buttons associated with the one the user has just clicked on.

38.3.13.2 wRadioSetButtonLT_t

A `wRadioSetButtonLT_t` is identical to a `wRadioSetButton_t` object except it draws the caption text to the left of the button rather than to the right of the button.

```
wRadioSetButtonLT_t:
    class inherits( wCheckable_t );

        procedure create_wRadioSetButtonLT
        (
            wrbltName   :string;
            caption     :string;
            style       :dword;
            parent      :dword;
            x           :dword;
            y           :dword;
            width       :dword;
            height      :dword;
            onClick     :widgetProc
        );  external;


    endclass;
```

`create_wRadioSetButtonLT`

This is the consructor for the `wRadioSetButtonLT_t` class. If you call this as a class procedure (e.g., "wRadioSetButtonLT_t.create_wRadioSetButtonLT") then this procedure will allocate storage for a new `wRadioSetButton_t` object on the heap and return a pointer to that object in ESI. If you make a standard object call to this constructor, then `create_wRadioSetButtonLT` will initialize that object in-place.

wrbName: HOWL assigns this string to the _name data field of the object. This string's value should be constant over the execution lifetime of the newly initialized object.

caption: this is the caption text that will be drawn to the left of the radio button. Windows makes an internal copy of this string, so the value need only exist for as long as the constructor call is in progress.

parent: this is the handle of the `wView_t` or `wForm_t` object on which the radio button will be drawn.

x, y, width, height: These arguments form a bounding box in which the radio button and caption will be drawn.

onClick: this is the name of a `widgetProc` that HOWL will call whenever you click on the radio button widget. If this field contains NULL, HOWL will not call any `widgetProc` procedure. Generally, if you're using `wRadioButton_t` objects in your application, it is the responsibility of the `onClick` procedure to properly update the other radio buttons associated with the one the user has just clicked on.

38.3.14 Editors and Edit Boxes

The HOWL edit widgets allow users to enter passwords, single lines of text, or text documents (up to 32KB long). Users can cut and paste data between edit widgets and perform many other text-editing functions. Applications can select text from an edit widget, insert text into the widget, or extract text from the widget. Indeed, with just a little extra code, it's quite possible to create a fully-featured text editor using the HOWL edit widgets.

Perhaps the biggest limitation to these widgets is their 32K text limitation. A future version of HOWL will include an extended text editor that overcomes this limitation.

All of the HOWL edit widgets are subclasses of the `wabsEditBox_t` class and, therefore, inherit all the fields and methods from that abstract base class. You should take a moment to review that abstract base class before looking at the following derived class definitions.

38.3.14.1 wEditBox_t

A wEditBox_t object allows a user to enter a single string (a single line of text) from the keyboard.

```
wEditBox_t:
    class inherits( wabsEditBox_t );

        procedure create_wEditBox
        (
            webName     :string;
            initialTxt  :string;
            parent      :dword;
            x           :dword;
            y           :dword;
            width       :dword;
            height      :dword;
            style       :dword;
            onChange    :widgetProc
        );  external;


    endclass;
```

`create_wEditBox` This is the consructor for the `wEditBox_t` class. If you call this as a class procedure (e.g., "wEditBox_t.create_wEditBox") then this procedure will allocate storage for a new `wEditBox_t` object on the heap and return a pointer to that object in ESI. If you make a standard object call to this constructor, then `create_wEditBox` will initialize that object in-place.

webName: HOWL assigns this string to the _name data field of the object. This string's value should be constant over the execution lifetime of the newly initialized object.

initialTxt: HOWL will initialize the edit box's text entry field with this string. This is commonly an empty string in most objects.

parent: this is the handle of the `wView_t` or `wForm_t` object on which the edit box will be drawn.

x, y, width, height: These arguments form a bounding box in which the edit box will be drawn.

style: This argument is zero or more of the Windows edit box styles logically OR'd together (or zero, to use the default edit box style). See the discussion of Windows edit box styles in the section on the `wEditBox` object earlier in this document (in the HOWL declaration language description).

onChange: this is the name of a `widgetProc` that HOWL will call whenever you change any text in the edit box widget. If this field contains NULL, HOWL will not call any `widgetProc` procedure. Generally, this field will contain NULL and you will process the text in an edit box in response to some other system event (such as a button press or loss of focus).

38.3.14.2 wPasswdBox_t

A `wPasswdBox_t` object is almost identical to a `wEditBox_t` object. The difference is that Windows substitutes asterisks (or some other user-defined character) for the characters the user types at the keyboard to protect passwords from prying eyes.

```
wPasswdBox_t:
    class inherits( wabsEditBox_t );

        procedure create_wPasswdBox
        (
            wpwbName    :string;
            initialTxt  :string;
            parent      :dword;
            x           :dword;
            y           :dword;
            width       :dword;
            height      :dword;
            style       :dword;
            onChange    :widgetProc
        );  external;


        method get_passwordChar; @returns( "eax" ); external;
        method set_passwordChar( pwc:char );        external;

    endclass;
```

`create_wPasswdBox`

This is the consructor for the `wPasswdBox_t` class. If you call this as a class procedure (e.g., "wPasswdBox_t.create_wPasswdBox") then this procedure will allocate storage for a new `wPasswdBox_t` object on the heap and return a pointer to that object in ESI. If you make a standard object call to this constructor, then `create_wPasswdBox` will initialize that object in-place.

webName: HOWL assigns this string to the _name data field of the object. This string's value should be constant over the execution lifetime of the newly initialized object.

initialTxt: HOWL will initialize the password box's text entry field with this string. This is commonly an empty string in most objects.

parent: this is the handle of the `wView_t` or `wForm_t` object on which the password box will be drawn.

x, y, width, height: These arguments form a bounding box in which the password box will be drawn.

style: This argument is zero or more of the Windows edit box styles logically OR'd together (or zero, to use the default edit box style). See the discussion of Windows edit box styles in the section on the `wEditBox` object earlier in this document (in the HOWL declaration language description).

onChange: this is the name of a `widgetProc` that HOWL will call whenever you change any text in the password box widget. If this field contains NULL, HOWL will not call any `widgetProc` procedure. Generally, this field will contain NULL and you will process the text in a password box in response to some other system event (such as a button press or loss of focus).

38.3.14.3 wTextEdit_t

A wTextEdit_t object allows the user to enter multiple lines of text in a text editor format. If the text editor string data contains more lines (or more characters on a given line) than will fit in the text editor window, Windows will automatically attach scroll bars to the window so the user can scroll through the text data.

```
wTextEdit_t:
    class inherits( wabsEditBox_t );

        procedure create_wTextEdit
        (
            wteName     :string;
            initialTxt  :string;
            parent      :dword;
            x           :dword;
            y           :dword;
            width       :dword;
            height      :dword;
            style       :dword;
            onChange    :widgetProc
        );  external;


        method getLineCount; @returns( "eax" );             external;

        method getLineIndex( charIndex:dword );
                    @returns( "eax" );                       external;

        method getCharIndex( lineIndex:dword );
                    @returns( "eax" );                       external;

        method getLine( lineIndex:dword; txt:string );       external;

        method a_getLine( lineIndex:dword );
                    @returns( "eax" );                       external;

        method scroll( horz:int32; vert:int32 );             external;
        method scrollCaret;                                  external;
        method setTabStops( tabstops:dword );                external;

    endclass;

create_wTextEdit
```

This is the consructor for the `wTextEdit_t` class. If you call this as a class procedure (e.g., "wTextEdit_t.create_wTextEdit") then this procedure will allocate storage for a new `wTextEdit_t` object on the heap and return a pointer to that object in ESI. If you make a standard object call to this constructor, then `create_wTextEdit` will initialize that object in-place.

webName: HOWL assigns this string to the _name data field of the object. This string's value should be constant over the execution lifetime of the newly initialized object.

initialTxt: HOWL will initialize the text editor's text entry field with this string. This is commonly an empty string in most objects.

parent: this is the handle of the `wView_t` or `wForm_t` object on which the text editor will be drawn.

x, y, width, height: These arguments form a bounding box in which the text editor will be drawn.

style: This argument is zero or more of the Windows edit box styles logically OR'd together (or zero, to use the default edit box style). See the discussion of Windows edit box styles in the section on the `wEditBox` object earlier in this document (in the HOWL declaration language description).

onChange: this is the name of a `widgetProc` that HOWL will call whenever you change any text in the text editor widget. If this field contains NULL, HOWL will not call any `widgetProc` procedure. Generally, this field will contain NULL and you will process the text in a text editor widget in response to some other system event (such as a button press or loss of focus).

`get_lineCount` This method returns the number of lines of text in the text editor widget.

`get_lineIndex` Given a zero-based character index into the text editor's string (up to 32K), this function will return a zero-based line number for that index (that is, the line number of the line that contains that particular character).

`get_charIndex` Given a (zero-based) line number into the text editor's data string, this function returns the (zero-based) character index (into the text editor's string data) of the first character on that line.

`get_line` Given a line index into the text editor's data string, this function returns the specified string.

lineIndex This is the zero-based line index into the text editor's data string.

txt: this is a string object wher get_line will copy the string data for the specified line. This string must be previously allocated and have sufficient storage to hold the string or HOWL will raise an exception.

`a_get_line` Given a line index, this method makes a copy of the specified text editor line on the heap and returns a pointer to this string in the EAX register. It is the caller's responsibility to free the storage associated with this string with the application is done using it.

`scroll` This function will scroll the text editor window the number of characters specified by the two arguments in the horizontal and vertical directions.

horz: this is the number of characters to scroll in the horizontal direction.

vert: this is the number of characters to scroll in the vertical direction. Windows will not let you scroll beyond the last line in the text editor's string; if you attempt to do so, Windows will simply display the last line of text at the top of the editor's window.

`scrollCaret` Positions the text display window so that the text containing the insertion caret is visible on the screen.

`setTabStops` This function sets tabstops every 'tabstops' characters, where 'tabstops' is the argument you pass to this method.

38.3.15 List, Drag, and Combo Boxes

List and drag boxes are tables of data from which the user can select an item (a row) by clicking on the line of text associated with that item. Applications can add, delete, or rearrange lines of text in list and drag boxes. End users can rearrange data in a drag box with no interaction from the application.

38.3.15.1 wListBox_t

A wListBox_t object is a table of strings created by the application. The user can click or double-click on these strings. The application can insert and delete strings in the list box. If there are too many strings to display in the window, then Windows will attach a vertical scroll bar to the list box and allow the user to scroll through the list box entry.

A list box will either display the strings in the order the application inserts them into the list box, or it can display them in a sorted order. You specify whether you want a sorted or unsorted list box when you create it.

```
wListBox_t:
    class inherits ( wClickable_t );

        var
            align( 4 );
            wListBox_private:
                record

                    textColor   :dword;

                endrecord;

        procedure create_wListBox
        (
            wlbName     :string;
            parent      :dword;
            x           :dword;
            y           :dword;
            width       :dword;
            height      :dword;
            sort        :boolean;
            onClick     :widgetProc
        );  external;

        method add_string( s:string );                      external;
        method insert_string( index:dword; s:string );      external;
        method delete_string( index:dword );                external;
        method reset;                                       external;

        method find_prefix
        (
            s           :string;
            startIndex  :dword
        );  @returns( "eax" );                              external;

        method find_string
        (
            s           :string;
            startIndex  :dword
        );  @returns( "eax" );                              external;

        method get_count;               @returns( "eax" );  external;
        method get_curSel;              @returns( "eax" );  external;
        method get_itemData( i:dword ); @returns( "eax" );  external;
        method a_get_text( i:dword );   @returns( "eax" );  external;
        method get_text( i:dword; s:string );               external;
```

```
            method set_curSel(index:dword); @returns( "eax" );  external;
            method set_itemData
            (
                index   :dword;
                data    :dword
            );  external;

            method load_dir
            (
                pathname    :string;
                attributes  :dword
            );  external;

            method get_textColor;   @returns( "eax" );          external;
            method set_textColor( textColor:dword );            external;

            override method processMessage;                     external;

        endclass;
```

`textColor` This is the RGB color that Windows will use to draw the text on the listbox. This is a private data field; applications should only access this value using the associated access and mutator. The constructor initializes the text color to black.

`create_wListBox`

This is the consructor for the `wListBox_t` class. If you call this as a class procedure (e.g., "wListBox_t.create_wListBox") then this procedure will allocate storage for a new `wListBox_t` object on the heap and return a pointer to that object in ESI. If you make a standard object call to this constructor, then `create_wListBox` will initialize that object in-place.

wlbName: HOWL assigns this string to the _name data field of the object. This string's value should be constant over the execution lifetime of the newly initialized object.

parent: this is the handle of the `wView_t` or `wForm_t` object on which the list box will be drawn.

x, y, width, height: These arguments form a bounding box in which the list box will be drawn.

sort: if this field is true, then the constructor will create a sorted list box. If this value is false, then the constructor will create an unsorted list box.

onClick: this is the name of a `widgetProc` that HOWL will call whenever select a (new) line in a list box.

`add_string` This method appends a string (specified as the argument) to the end of an unsorted list box. It will insert the string at the proper position within a sorted list box. Note that Windows will make an internal copy of the string's data.

`insert_string` This method inserts a string before some other entry in the list box. This method ignores the sorted/unsorted state of the list box and always inserts the string at the specified index. The index must be in the range 0..count where count is the number of entries in the list box. If you specify the value count as the index, then this method appends the item to the end of the list (similar to `add_string`, except no sorting).

index: the line number before which the string is to be inserted. This is a zero-based index.

s: this is the string data to insert into the list box.

`delete_string` This method deletes the string at the specified index in the list box.

`reset` This method deletes all the strings in the list box.

`find_prefix` This method searches for a line of text in the list box that begins with some string. This method begins searching starting with an application-defined line index into the list box.

This method returns the index into the list box where the string prefix was found, or the constant w.LB_ERR if it could find no string with the specified prefix.

s: the string prefix to search for in the list box.

startIndex: the starting line index to begin the search.

`find_string` This method searches for a line of text in the list box that matches some string. This method begins searching starting with an application-defined line index into the list box. This method returns the index into the list box where the string was found, or the constant w.LB_ERR if it could not find the string.

s: the string to search for in the list box.

startIndex: the starting line index to begin the search.

`get_count` This method returns the number of lines in the list box (in EAX).

`get_curSel` This method returns the index of the currently selected item in the list box (in EAX).

`get_itemData` Each line of text in a list box has a 32-bit user-defined data value associated with it. You could, for example store a pointer to some object or other data type in this field and retrieve it when the user selects an item in the list box. The `get_itemData` method retrieves this user data from the list box. The single argument is the index of the list box entry for which you want the user data (you would typically supply the data returned by `get_curSel` as this argument).

`a_get_text` This method makes a copy of the string data (on the heap) for the line in the list box at the index specified by the argument. It returns a pointer to this new string in EAX. It is the caller's responsibility to free the storage for this string when the caller is done with it.

`get_text` This method makes a copy of the string data for the line in the list box at a user-supplied index. It stores the string data into a string object whose address the caller passes as an argument. That string must have sufficient storage allocated for it or HOWL will raise an exception.

i: index of the line in the list box whose string data this method will extract.

s: pointer to a string object where this method will store the result.

`set_curSel` This methods highlights the line at the specified index in the list box (it becomes the "currently selected" item).

`set_itemData` This method allows you to associate a user-defined 32-bit data value with an item in the list box. If 32 bits is insufficient for your needs, you can always store a pointer to the actual data in this data area.

index: this is the (zero-based) index of the line in the list box that you want to attach the data to.

data: this is the 32-bit value you want to associate with the line in the list box.

`load_dir` This method populates the list box with the file names from the directory specified by the `pathname` argument.

pathname: an ambiguous pathname (e.g., "c:*.*" that specifies the path to the files and the files at that path tha t you want to load into the list box.

attributes: either zero, or the logical OR of one or more of the following Windows' attribute constants:

`w.DDL_ARCHIVE`	Includes archived files.
`w.DDL_DIRECTORY`	Includes subdirectories. Subdirectory names are enclosed in square brackets ([ ]).
`w.DDL_DRIVES`	Includes drives. Drives are listed in the form [-x-], where x is the drive letter.
`w.DDL_EXCLUSIVE`	Includes only files with the specified attributes. By default, read-write files are listed even if DDL_READWRITE is not specified.

w.DDL_HIDDEN	Includes hidden files.
w.DDL_READONLY	Includes read-only files.
w.DDL_READWRITE	Includes read-write files with no additional attributes.
w.DDL_SYSTEM	Includes system files.

get_textColor,	
set_textColor	These accessor/mutator functions get and set the text color that the widget uses.
processMessage	This is a private method. Applications must not call it.

38.3.15.2 wDragListBox_t

A wDragListBox_t object is a special kind of list box that allows the user to rearrange the items in the list box without any interaction from the application. An application uses drag list boxes exactly like list boxes (except, of course, for the object's type name). As the wDragListBox_t class is derived from wListBox_t, all of the list box methods are available to wDragListBox_t objects. Note that HOWL does not offer drag list boxes the "sort" option, which makes little sense as the end user will probably rearrange their drag list boxes thus defeating the purpose of the sort option.

```
wDragListBox_t:
    class inherits ( wListBox_t );

        var
            align( 4 );
            wDragListBox_private:
                record

                    // The following is a private field.
                    // External code should not access it.

                    startDragIndex :dword;

                endrecord;

        procedure create_wDragListBox
        (
            wlbName     :string;
            parent      :dword;
            x           :dword;
            y           :dword;
            width       :dword;
            height      :dword;
            onClick     :widgetProc
        );  external;

        override method processMessage;                    external;

    endclass;
```

create_wDragListBox

This is the consructor for the wDragListBox_t class. If you call this as a class procedure (e.g., "wDragListBox_t.create_wDragListBox") then this procedure will allocate storage for a new wDragListBox_t object on the heap and return a pointer to that object in ESI. If you make a standard object call to this constructor, then create_wDragListBox will initialize that object in-place.

wlbName: HOWL assigns this string to the _name data field of the object. This string's value should be constant over the execution lifetime of the newly initialized object.

parent: this is the handle of the `wView_t` or `wForm_t` object on which the drag list box will be drawn.

x, y, width, height: These arguments form a bounding box in which the drag list box will be drawn.

onClick: this is the name of a `widgetProc` that HOWL will call whenever the user selects a (new) line in a drag list box.

38.3.15.3 wComboBox_t

The `wComboBox_t` object is a combination of an edit box, a list box, and a pull-down menu. The user can type text directly into a list box or click on a button attached to the combo box and select an item from a list box that appears in a pull-down menu.

```
wComboBox_t:
    class inherits ( wListBox_t );

        var
            align( 4 );
            wComboBox_private:
                record

                    onEditChange    :widgetProc;
                    onCancel        :widgetProc;
                    onSelEndOk      :widgetProc;

                endrecord;

        procedure create_wComboBox
        (
            wcbName     :string;
            caption     :string;
            parent      :dword;
            x           :dword;
            y           :dword;
            width       :dword;
            height      :dword;
            sort        :boolean;
            onSelChange :widgetProc
        );  external;

        method get_onEditChange;    @returns( "eax" );      external;
        method get_onCancel;        @returns( "eax" );      external;
        method get_SelEndOk;        @returns( "eax" );      external;

        method set_onEditChange( onEditChange:widgetProc ); external;
        method set_onCancel( onCancel:widgetProc );         external;
        method set_SelEndOk( onSelEndOk:widgetProc );       external;

        method a_get_editBoxText;   @returns( "eax" );      external;
        method get_editBoxText( theText:string );           external;

        method set_editBoxText( theText:string );           external;


        override method load_dir;                           external;
        override method processMessage;                     external;
```

```
            override method add_string;                     external;
            override method insert_string;                  external;
            override method delete_string;                  external;
            override method reset;                          external;
            override method find_prefix;                    external;
            override method find_string;                    external;
            override method get_count;                      external;
            override method get_curSel;                     external;
            override method get_itemData;                   external;
            override method a_get_text;                     external;
            override method get_text;                       external;
            override method set_curSel;                     external;
            override method set_itemData;                   external;

        endclass;
```

onEditChange — This is a pointer to a widgetProc that HOWL will call whenever the user makes a change to the edit box component of a combo box. If this field is NULL, HOWL will ignore it. Note that applications should only access or modify this field using the associated accessor/mutator methods.

onCancel — This is a pointer to a widgetProc that HOWL will call whenever the user cancels a change to the edit box component of a combo box. If this field is NULL, HOWL will ignore it. Note that applications should only access or modify this field using the associated accessor/mutator methods.

onSelEndOk — This is a pointer to a widgetProc that HOWL will call whenever the user selects and item from the pull-down list box and the application should select that entry. If this field is NULL, HOWL will ignore it. Note that applications should only access or modify this field using the associated accessor/mutator methods.

`create_wComboBox`

This is the consructor for the `wComboBox_t` class. If you call this as a class procedure (e.g., "wComboBox_t.create_wComboBox") then this procedure will allocate storage for a new `wComboBox_t` object on the heap and return a pointer to that object in ESI. If you make a standard object call to this constructor, then `create_wComboBox` will initialize that object in-place.

wcbName: HOWL assigns this string to the _name data field of the object. This string's value should be constant over the execution lifetime of the newly initialized object.

caption: This is the initial string data for the combo box's edit box field.

parent: this is the handle of the `wView_t` or `wForm_t` object on which the combo box will be drawn.

x, y, width, height: These arguments form a bounding box in which the combo box will be drawn.

sort: like list boxes, combo boxes offer the option of sorting the list for you. If this field is true, HOWL will create a sorted list in the combo box; if this field is false, the list in the combo box will be unsorted.

onSelChange: this is the name of a `widgetProc` that HOWL will call whenever the user selects a (new) line in the list of a combo box.

`get_onEditChange`,
`get_onCancel`,
`get_onSelEndOk` — These are the accessor functions for the combo box's data fields.
`set_onEditChange`,
`set_onCancel`,

`set_onSelEndOk`	These are the mutator functions for the combo box's data fields.
a_get_editBox_text	This method returns a copy of the string data currently held in the combo box's edit box component. HOWL allocates storage for this string and returns a pointer to the new string in the EAX register. It is the caller's responsibility to free the storage for this string.
get_editBox_text	This method retrieves the string from the combo box and stores the string data in the string object passed as a parameter. The destination string must have sufficient storage or HOWL will raise an exception.
set_editBox_text	This function replaces the combo box's string data with the string passed as an argument.

The remaining methods listed in the `wComboBox_t` declaration above are overriden methods from the `wListBox_t` class. The overriding occurs for internal technical reasons. To an application, these methods are used exactly like those in a list box. Please see the discussion in the list box section for more details on the operation of these methods.

38.3.16 Progress Bars

A progress bar is a bar graph that shows the progress of some lengthy operation during the execution of an application.

38.3.16.1 wProgressBar_t

The `wProgressBar_t` type implements Windows progress bars in HOWL. A progress bar has three main attributes: a current position (the current "progress"), a minimum position, and a maximum position. When drawing a progress bar, Windows will create a horizontal bar graph and will fill the bar graph from the left to the right based on the current position, minimum, and maximum values.

```
wProgressBar_t:
    class inherits( wVisual_t );

        var
            align( 4 );
            wProgressBar_private:
                record

                    position    :word;
                    align( 4 );

                    lowRange    :word;
                    hiRange     :word;

                endrecord;

        procedure create_wProgressBar
        (
            wpbName     :string;
            parent      :dword;
            x           :dword;
            y           :dword;
            width       :dword;
            height      :dword
        );  external;

        method get_position; @returns( "eax" );         external;
        method set_position( position:word );           external;

        method get_lowRange; @returns( "eax" );         external;
        method get_hiRange;  @returns( "eax" );         external;
        method set_range( low:word; high:word );        external;
```

```
        endclass;
```

position	These data field is the current position of the track bar. Applications should never access this field directly. Instead, they should use the accessor/mutator methods to read or write this field's data.
lowRange	This is the minimum value for the progress bar's position. It is usually zero, but you can set any minimum value you like as long as it is less than the value held in the `hiRange` data field. You should never set the value of the `position` data field to a value lower than the `lowRange` value. Applications should never access this field directly. Instead, they should use the accessor/mutator methods to read or write this field's data.
hiRange	This is the maximum value for the progress bar's position. You can set any maximum value you like as long as it is greater than the value held in the `lowRange` data field. You should never set the value of the `position` data field to a value higher than the `hiRange` value. Applications should never access this field directly. Instead, they should use the accessor/mutator methods to read or write this field's data.

`create_wProgressBar`

This is the consructor for the `wProgressBar_t` class. If you call this as a class procedure (e.g., "wProgressBar_t.create_wProgressBar") then this procedure will allocate storage for a new `wProgressBar_t` object on the heap and return a pointer to that object in ESI. If you make a standard object call to this constructor, then `create_wProgressBar` will initialize that object in-place.

wpbName: HOWL assigns this string to the _name data field of the object. This string's value should be constant over the execution lifetime of the newly initialized object.

parent: this is the handle of the `wView_t` or `wForm_t` object on which the progress bar will be drawn.

x, y, width, height: These arguments form a bounding box in which the progress bar will be drawn.

`get_position`,
`get_lowRange`,
`get_hiRange` These are the accessor methods for the class' data fields.

`set_position`,
`set_lowRange`,
`set_hiRange` These are the mutator functions for the class' data fields. Note that changing any of these values (and in particular, changing the position value) will cause Windows to redraw the progress bar reflecting the new values.

38.3.17 Scroll Bars and Track Bars

Scroll bars and track bars are positional input devices. Applications generally use scroll bars to specify a position to view on a form (when the form's contents are too large to fit in the window and one time) whereas a trackbar provides a generic numeric input device selectable by the position of the slider on the trackbar. Both widgets are available in horizontal and vertical orientations.

Note that Windows can automatically associate horizontal and vertical scroll bars with a window when you create that window. For window scrolling purposes, this is generally how you create and use scroll bars. However, you can create stand-alone scroll bars for use by your applications using the scrollbar widget.

38.3.17.1 wScrollBar_t

The wScrollBar_t class lets you create stand-alone scroll bars in your application. Scroll bars offer a large number of event notifications that tell you about various user interactions with the scroll bar.

Scroll bars in early versions of Windows were limited to 16-bit range and position values. For the most part, later versions of Windows extended all these values to 32 bits. However, one important pair of values, returned when tracking movements on scroll bars, is still limited to 16-bit values. Therefore, if you plan to make full use of the scroll bar's feature set and notifications, you need to limit yourself to using 16-bit ranges and positions. In general, this is not a severe limitations because there aren't enough pixels on the screen to provide a granularit of 16 bits, much less 32. However, just note that although you can set the low and high range values for a scroll bar to arbitrary 32-bit values, you should limit yourself to 16-bit values.

```
wScrollBar_t:
    class inherits( wVisual_t );

        var
            align( 4 );
            wScrollBar_private:
                record

                    onChange        :widgetProc;
                    onThumbPosn     :widgetProc;
                    onThumbTrack    :widgetProc;
                    onLineLeft      :widgetProc;
                    onLineRight     :widgetProc;
                    onLineDown      :widgetProc;
                    onLineUp        :widgetProc;
                    onEndScroll     :widgetProc;
                    onPageDown      :widgetProc;
                    onPageUp        :widgetProc;
                    onPageLeft      :widgetProc;
                    onPageRight     :widgetProc;
                    onTop           :widgetProc;
                    onBottom        :widgetProc;

                    lineInc         :uns32;
                    pageInc         :uns32;
                    curPosn         :dword;
                    info            :w.SCROLLINFO;
                    textColor       :dword;

                endrecord;

            procedure create_wScrollBar
            (
                wtbName     :string;
                parent      :dword;
                x           :dword;
                y           :dword;
                width       :dword;
                height      :dword;
                style       :dword;
                onChange    :widgetProc
            );  external;

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

            method get_position; @returns( "eax" );             external;
            method set_position( position:dword );              external;

            method get_lowRange; @returns( "eax" );             external;
            method get_hiRange;  @returns( "eax" );             external;
```

```
        method set_range( low:dword; high:dword );          external;

        method get_onChange;        @returns( "eax" );      external;
        method get_onThumbPosn;     @returns( "eax" );      external;
        method get_onThumbTrack;    @returns( "eax" );      external;
        method get_onLineDown;      @returns( "eax" );      external;
        method get_onLineUp;        @returns( "eax" );      external;
        method get_onLineLeft;      @returns( "eax" );      external;
        method get_onLineRight;     @returns( "eax" );      external;
        method get_onEndScroll;     @returns( "eax" );      external;
        method get_onPageDown;      @returns( "eax" );      external;
        method get_onPageUp;        @returns( "eax" );      external;
        method get_onPageLeft;      @returns( "eax" );      external;
        method get_onPageRight;     @returns( "eax" );      external;
        method get_onTop;           @returns( "eax" );      external;
        method get_onBottom;        @returns( "eax" );      external;
        method get_lineInc;         @returns( "eax" );      external;
        method get_pageInc;         @returns( "eax" );      external;

        method set_onChange( onChange:widgetProc );         external;
        method set_onThumbPosn( onThumbPosn:widgetProc );   external;
        method set_onThumbTrack( onThumbTrack:widgetProc ); external;
        method set_onLineDown( onLineDown:widgetProc );     external;
        method set_onLineUp( onLineUp:widgetProc );         external;
        method set_onLineLeft( onLineLeft:widgetProc );     external;
        method set_onLineRight( onLineRight:widgetProc );   external;
        method set_onEndScroll( onEndScroll:widgetProc );   external;
        method set_onPageDown( onPageDown:widgetProc );     external;
        method set_onPageUp( onPageUp:widgetProc );         external;
        method set_onPageLeft( onPageLeft:widgetProc );     external;
        method set_onPageRight( onPageRight:widgetProc );   external;
        method set_onTop( onTop:widgetProc );               external;
        method set_onBottom( onBottom:widgetProc );         external;

        method set_lineInc( lineInc:dword );                external;
        method set_pageInc( pageInc:dword );                external;

        method get_textColor;   @returns( "eax" );          external;

        method set_textColor( textColor:dword );            external;

        override method processMessage;                     external;


    endclass;
```

`onChange` This field, if non-NULL, points at a `widgetProc` procedure that HOWL will call whenever the user changes the position of the thumb on the scroll bar (using any means to change the value). Upon entry into the `widgetProc` procedure, the H.O. word of the wParam argument contains the current scroll bar position.

`onThumbPosn` This field, if non-NULL, points at a `widgetProc` procedure that HOWL will call whenever the user changes the position of the thumb on the scroll bar by dragging the thumb. HOWL calls this procedure after the user has released the mouse button while dragging the thumb control. Upon entry into the `widgetProc` procedure, the H.O. word of the wParam argument contains the current scroll bar position.

`onThumbTrack` This field, if non-NULL, points at a `widgetProc` procedure that HOWL will call while the user is dragging the thumb around the scroll bar. An application can use this

notification to dynamically adjust the screen during thumb movement. Upon entry into the `widgetProc` procedure, the H.O. word of the wParam argument contains the current scroll bar position. Note that this is the only notification that provides only a 16-bit value for the thumb position; if you don't need to use this notification, it is possible to obtain 32-bit values for the thumb position (from the other notification calls).

`onLineLeft` This field, if non-NULL, points at a `widgetProc` procedure that HOWL will call whenever the user changes the position of the thumb on a horizontal scroll bar by pressing the left arrow on the scroll bar. Upon entry into the `widgetProc` procedure, the H.O. word of the wParam argument contains the current scroll bar position.

`onLineRight` This field, if non-NULL, points at a `widgetProc` procedure that HOWL will call whenever the user changes the position of the thumb on a horizontal scroll bar by pressing the right arrow on the scroll bar. Upon entry into the `widgetProc` procedure, the H.O. word of the wParam argument contains the current scroll bar position.

`onLineDown` This field, if non-NULL, points at a `widgetProc` procedure that HOWL will call whenever the user changes the position of the thumb on a vertical scroll bar by pressing the down arrow on the scroll bar. Upon entry into the `widgetProc` procedure, the H.O. word of the wParam argument contains the current scroll bar position.

`onLineUp` This field, if non-NULL, points at a `widgetProc` procedure that HOWL will call whenever the user changes the position of the thumb on a vertical scroll bar by pressing the up arrow on the scroll bar. Upon entry into the `widgetProc` procedure, the H.O. word of the wParam argument contains the current scroll bar position.

`onEndScroll` This field, if non-NULL, points at a `widgetProc` procedure that HOWL will call after any scroll operation is complete. Upon entry into the `widgetProc` procedure, the H.O. word of the wParam argument contains the current scroll bar position.

`onPageDown` This field, if non-NULL, points at a `widgetProc` procedure that HOWL will call whenever the user changes the position of the thumb on a vertical scroll bar by clicking on the scroll bar between the thumb and the down arrow on the scroll bar. Upon entry into the `widgetProc` procedure, the H.O. word of the wParam argument contains the current scroll bar position.

`onPageUp` This field, if non-NULL, points at a `widgetProc` procedure that HOWL will call whenever the user changes the position of the thumb on a vertical scroll bar by clicking on the scroll bar between the thumb and the up arrow on the scroll bar. Upon entry into the `widgetProc` procedure, the H.O. word of the wParam argument contains the current scroll bar position.

`onPageLeft` This field, if non-NULL, points at a `widgetProc` procedure that HOWL will call whenever the user changes the position of the thumb on a horizontal scroll bar by clicking on the scroll bar between the thumb and the left arrow on the scroll bar. Upon entry into the `widgetProc` procedure, the H.O. word of the wParam argument contains the current scroll bar position.

`onPageRight` This field, if non-NULL, points at a `widgetProc` procedure that HOWL will call whenever the user changes the position of the thumb on a horizontal scroll bar by clicking on the scroll bar between the thumb and the right arrow on the scroll bar. Upon entry into the `widgetProc` procedure, the H.O. word of the wParam argument contains the current scroll bar position.

`onTop` This field, if non-NULL, points at a `widgetProc` procedure that HOWL will call whenever the user moves the thumb all the way to the top on a vertical scroll bar or all the way to the left on a horizontal scroll bar. Upon entry into the `widgetProc` procedure, the H.O. word of the wParam argument contains the current scroll bar position (which should be the maximum value).

`onBottom` This field, if non-NULL, points at a `widgetProc` procedure that HOWL will call whenever the user moves the thumb all the way to the bottom on a vertical scroll bar or all the way to the right on a horizontal scroll bar. Upon entry into the `widgetProc` procedure, the H.O. word of the wParam argument contains the current scroll bar position (which should be the maximum value).

`lineInc` This data field contains the number of positions that will be added to or subtracted from the scroll bar thumb's current position when the user clicks on one of the scroll bar's arrow buttons. Applications should not access this data field directly, they should use the standard class accessor/mutator functions to read and write this value.

`pageInc` This data field contains the number of positions that will be added to or subtracted from the scroll bar thumb's current position when the user clicks on the scroll bar between the thumb and one of the arrow buttons. Applications should not access this data field directly, they should use the standard class accessor/mutator functions to read and write this value.

`curPosn` This is a private field used by the class. Applications should not access it.

`info` This is a private field used by the class. Applications should not access it.

`textColor` This is the RGB color that Windows will use to draw the text on the scrollbar. This is a private data field; applications should only access this value using the associated access and mutator. The constructor initializes the text color to black.

`bkgColor` This is the RGB color that Windows will use to paint the background of the scrollbar. This is a private data field; applications should only access this value using the associated access and mutator. The constructor initializes the backgroun color to white.

`bkgBrush` This is a private data field; applications should never access it.

`create_wScrollBar`

This is the consructor for the `wScrollBar_t` class. If you call this as a class procedure (e.g., "wScrollBar_t.create_wScrollBar") then this procedure will allocate storage for a new `wScrollBar_t` object on the heap and return a pointer to that object in ESI. If you make a standard object call to this constructor, then `create_wScrollBar` will initialize that object in-place.

wtbName: HOWL assigns this string to the _name data field of the object. This string's value should be constant over the execution lifetime of the newly initialized object.

parent: this is the handle of the `wView_t` or `wForm_t` object on which the scroll bar will be drawn.

x, y, width, height: These arguments form a bounding box in which the scroll bar will be drawn.

style: HOWL logically ORs this value with the Windows styles (w.WS_CHILD | w.WS_VISIBLE) when creating the scroll bar. At the very least, this field should contain w.SBS_HORZ for a horizontal scroll bar or w.SBS_VERT for a vertical scroll bar. Please see the Windows documentation or the HLA *w.hhf* header file for additional scroll bar style (SBS_*) constants.

onChange: this is the address (which can be NULL) of an onChange widgetProc procedure. The constructor will initialize the onChange field with this value.

`enable`, `disable` These two methods will enable or disable the scroll bar on the form. A disabled scroll bar is still visible, but the user will be unable to interact with it.

`show`, `hide` These two methos will make a scroll bar visible or invisible on the form.

`get_position` This function returns the current scroll bar position as a 32-bit value in the EAX register. Note that you can call this method from any of the event notification procedures except `onThumbTrack` to obtain the true 32-bit position of the thumb control (rather than limiting yourself to the upper 16 bits of the `wParam` parameter). If you call this method from the `onThumbTrack` notification procedure, it will not return the current position of the thumb, instead it will return the last position before the user started dragging the thumb around. Sadly, there is no way to obtain a 32-bit current thumb position while dragging the thumb.

`set_position` This method sets the current thumb position on the scroll bar. The argument you pass to this method must be between the low and high range values for the scroll bar.

`get_lowRange` This method returns (in EAX) the current lower range for the scroll bar. This is typically zero, but you can program any 32-bit value you desire (see `set_range`).

`get_hiRange` This method returns (in EAX) the current upper range for the scroll bar.

`set_range` This method lets you set the low and high range values for the scroll bar. The default range is 0..100, but you can set any 32-bit values you like. Note that if you intend to use the `onThumbTrack` notification, you should limit the range to 16-bit values.

low: the lower bound of the scroll bar range

high: the upper bound of the scroll bar range.

`get_onChange`,
`get_onThumbPosn`,
`get_onThumbTrack`,
`get_onLineDown`,
`get_onLineUp`,
`get_onLineLeft`,
`get_onLineRight`,
`get_onEndScroll`,
`get_onPageDown`,
`get_onPageUp`,
`get_onPageLeft`,
`get_onPageRight`,
`get_onTop`,
`get_onBottom` These are the accessor methods for the corresponding `widgetProc` pointer data fields in this class.

`get_lineInc`,
`get_pageInc` These are the accessor methods for the corresponding data fields in the class. Applications should always use these accessor methods to read the values of the `lineInc` and `pageInc` fields.

set_onChange,
set_onThumbPosn,
set_onThumbTrack,
set_onLineDown,
set_onLineUp,
set_onLineLeft,
set_onLineRight,
set_onEndScroll,
set_onPageDown,
set_onPageUp,
set_onPageLeft,
set_onPageRight,
set_onTop,
set_onBottom These are the mutator functions for all the widgetProc pointer fields in the class.
`set_lineInc`,

`set_pageInc`	These are the mutator methods for the corresponding data fields in the class. Applications should always use these accessor methods to write the values of the `lineInc` and `pageInc` fields.
`get_textColor`, `set_textColor`, `get_bkgColor`, `set_bkgColor`	These accessor/mutator functions get and set the text and background colors that the widget uses.
`processMessage`	This is a private method in the class. Applications must not call this method.

38.3.17.2 wTrackBar_t

`wTrackBar_t` objects are very similar in use to a scroll bar insofar as they provide a slider control that the user can move between two extremes on the widget. However, whereas scroll bars have a specific user interface purpose (moving the view through a window), track bars are generalized numeric input devices. Their purpose is to input a numeric value between a low range and a high range via a slider control.

Like scroll bars, you should limit the range of trackbar responses to 16 bits if you intend to use the `onThumbTrack` notification.

```
wTrackBar_t:
    class inherits( wVisual_t );

        var
            align( 4 );
            wTrackBar_private:
                record

                    onChange        :widgetProc;
                    onThumbPosn     :widgetProc;
                    onThumbTrack    :widgetProc;
                    onBottom        :widgetProc;
                    onLineDown      :widgetProc;
                    onLineUp        :widgetProc;
                    onTop           :widgetProc;
                    onEndtrack      :widgetProc;
                    onPageDown      :widgetProc;
                    onPageUp        :widgetProc;

                endrecord;

        procedure create_wTrackBar
        (
            wtbName     :string;
            parent      :dword;
            x           :dword;
            y           :dword;
            width       :dword;
            height      :dword;
            style       :dword;
            onChange    :widgetProc
        );  external;

        method get_position; @returns( "eax" );             external;
        method set_position( position:dword );              external;

        method get_lowRange; @returns( "eax" );             external;
        method get_hiRange;  @returns( "eax" );             external;
        method set_range( low:dword; high:dword );          external;
```

```
            method get_onChange;        @returns( "eax" );      external;
            method get_onThumbPosn;     @returns( "eax" );      external;
            method get_onThumbTrack;    @returns( "eax" );      external;
            method get_onBottom;        @returns( "eax" );      external;
            method get_onLineDown;      @returns( "eax" );      external;
            method get_onLineUp;        @returns( "eax" );      external;
            method get_onTop;           @returns( "eax" );      external;
            method get_onEndtrack;      @returns( "eax" );      external;
            method get_onPageDown;      @returns( "eax" );      external;
            method get_onPageUp;        @returns( "eax" );      external;

            method set_onChange( onChange:widgetProc );         external;
            method set_onThumbPosn( onThumbPosn:widgetProc );   external;
            method set_onThumbTrack( onThumbTrack:widgetProc ); external;
            method set_onBottom( onBottom:widgetProc );         external;
            method set_onLineDown( onLineDown:widgetProc );     external;
            method set_onLineUp( onLineUp:widgetProc );         external;
            method set_onTop( onTop:widgetProc );               external;
            method set_onEndtrack( onEndtrack:widgetProc );     external;
            method set_onPageDown( onPageDown:widgetProc );     external;
            method set_onPageUp( onPageUp:widgetProc );         external;

            override method processMessage;                     external;


        endclass;
```

`onChange` This field, if non-NULL, points at a `widgetProc` procedure that HOWL will call whenever the user changes the position of the slider on the track bar (using any means to change the value). Upon entry into the `widgetProc` procedure, the H.O. word of the wParam argument contains the current track bar position.

`onThumbPosn` This field, if non-NULL, points at a `widgetProc` procedure that HOWL will call whenever the user changes the position of the slider on the track bar by dragging the slider. HOWL calls this procedure after the user has released the mouse button while dragging the slider control. Upon entry into the `widgetProc` procedure, the H.O. word of the wParam argument contains the current scroll bar position.

`onThumbTrack` This field, if non-NULL, points at a `widgetProc` procedure that HOWL will call while the user is dragging the slider around the track bar. An application can use this notification to dynamically adjust the system during slider movement. Upon entry into the `widgetProc` procedure, the H.O. word of the wParam argument contains the current track bar slider position.

`onBottom` This field, if non-NULL, points at a `widgetProc` procedure that HOWL will call whenever the user moves the slider all the way to the bottom on a vertical track bar or all the way to the left on a horizontal track bar. Upon entry into the `widgetProc` procedure, the H.O. word of the wParam argument contains the current track bar position (which should be the maximum value).

`onLineDown` This field, if non-NULL, points at a `widgetProc` procedure that HOWL will call whenever the user changes the position of the slider on a track bar by pressing the down arrow or the right arrow key on the keyboard. Upon entry into the `widgetProc` procedure, the H.O. word of the wParam argument contains the current track bar position.

`onLineUp` This field, if non-NULL, points at a `widgetProc` procedure that HOWL will call whenever the user changes the position of the slider on a track bar by pressing the up arrow or left arrow key on the keyboard. Upon entry into the `widgetProc` procedure, the H.O. word of the wParam argument contains the current track bar position.

`onTop` This field, if non-NULL, points at a `widgetProc` procedure that HOWL will call whenever the user moves the slider all the way to the top on a vertical track bar or all the way to the left on a horizontal track bar. Upon entry into the `widgetProc` procedure, the H.O. word of the wParam argument contains the current track bar position (which should be the maximum value).

`onEndTrack` This field, if non-NULL, points at a `widgetProc` procedure that HOWL will call after any trackbar operation is complete. Upon entry into the `widgetProc` procedure, the H.O. word of the wParam argument contains the current track bar position.

`onPageDown` This field, if non-NULL, points at a `widgetProc` procedure that HOWL will call whenever the user changes the position of the slider on a vertical scroll bar by clicking on the track bar between the slider and the left or bottom end of the track bar. Upon entry into the `widgetProc` procedure, the H.O. word of the wParam argument contains the current track bar position.

`onPageUp` This field, if non-NULL, points at a `widgetProc` procedure that HOWL will call whenever the user changes the position of the slider on a track bar by clicking on the track bar between the slider and the top or right side of the track bar. Upon entry into the `widgetProc` procedure, the H.O. word of the wParam argument contains the current track bar position.

`create_wTrackBar`

This is the consructor for the `wTrackBar_t` class. If you call this as a class procedure (e.g., "wTrackBar_t.create_wTrackBar") then this procedure will allocate storage for a new `wTrackBar_t` object on the heap and return a pointer to that object in ESI. If you make a standard object call to this constructor, then `create_wTrackBar` will initialize that object in-place.

wtbName: HOWL assigns this string to the _name data field of the object. This string's value should be constant over the execution lifetime of the newly initialized object.

parent: this is the handle of the `wView_t` or `wForm_t` object on which the track bar will be drawn.

x, y, width, height: These arguments form a bounding box in which the track bar will be drawn.

style: HOWL logically ORs this value with the Windows styles (w.WS_CHILD | w.WS_VISIBLE | w.TBS_AUTOTICKS) when creating the scroll bar. At the very least, this field should contain w.TBS_HORZ for a horizontal track bar or w.TBS_HORZ for a vertical track bar. Please see the Windows documentation or the HLA *w.hhf* header file for additional track bar style (TBS_*) constants.

onChange: this is the address (which can be NULL) of an `onChange` widgetProc procedure. The constructor will initialize the `onChange` field with this value.

`get_position` This function returns the current track bar position as a 32-bit value in the EAX register. Note that you can call this method from any of the event notification procedures except `onThumbTrack` to obtain the true 32-bit position of the slider control (rather than limiting yourself to the upper 16 bits of the `wParam` parameter). If you call this method from the `onThumbTrack` notification procedure, it will not return the current position of the slider, instead it will return the last position before the user started dragging the slider around. Sadly, there is no way to obtain a 32-bit current thumb position while dragging the thumb.

`set_position` This method sets the current thumb position on the track bar. The argument you pass to this method must be between the low and high range values for the track bar.

`get_lowRange` This method returns (in EAX) the current lower range for the track bar. This is typically zero, but you can program any 32-bit value you desire (see `set_range`).

`get_hiRange` This method returns (in EAX) the current upper range for the track bar.

`set_range`	This method lets you set the low and high range values for the track bar. The default range is 0..100, but you can set any 32-bit values you like. Note that if you intend to use the `onThumbTrack` notification, you should limit the range to 16-bit values. **low**: the lower bound of the track bar range **high**: the upper bound of the track bar range.
`get_onChange`, `get_onThumbPosn`, `get_onThumbTrack`, `get_onLineDown`, `get_onLineUp`, `get_onLineLeft`, `get_onLineRight`, `get_onEndtrack`, `get_onPageDown`, `get_onPageUp`, `get_onPageLeft`, `get_onPageRight`, `get_onTop`, `get_onBottom`	These are the accessor methods for the corresponding `widgetProc` pointer data fields in this class.
set_onChange, set_onThumbPosn, set_onThumbTrack, set_onLineDown, set_onLineUp, set_onLineLeft, set_onLineRight, set_onEndtrack, set_onPageDown, set_onPageUp, set_onPageLeft, set_onPageRight, set_onTop, set_onBottom	These are the mutator functions for all the widgetProc pointer fields in the class.
`processMessage`	This is a private method in the class. Applications must not call this method.

38.3.18 Up/Down Arrows

An up/down arrow control is a pair of (stacked) arrow buttons that that user can click on in order to increment or decrement a value. Up/down arrow widgets can be stand-alone or they can be attached to a "buddy" edit box widget (wUpDownEditBox_t widgets_. When attached to a buddy edit box, the up/down arrow control reflects the current value of the control in the edit box.

38.3.18.1 wUpDown_t

The `wUpDown_t` class is used to create up/down arrow objects on a form.

```
wUpDown_t:
    class inherits( wClickable_t );

        procedure create_wUpDown
        (
            wudName     :string;
            parent      :dword;
            alignment   :dword;
            x           :dword;
            y           :dword;
            width       :dword;
            height      :dword;
            lowerRange  :dword;
            upperRange  :dword;
            initialPosn :dword;
            onClick     :widgetProc
        );  external;

        method get_lowerRange;  @returns( "eax" );          external;
        method get_upperRange;  @returns( "eax" );          external;
        method get_position;    @returns( "eax" );          external;

        method set_lowerRange( lowerRange:word );           external;
        method set_upperRange( upperRange:word );           external;
        method set_position(   position  :word );           external;

        override method processMessage;                     external;

    endclass;
```

`create_wUpDown`

This is the consructor for the `wUpDown_t` class. If you call this as a class procedure (e.g., "wUpDown_t.create_wUpDown") then this procedure will allocate storage for a new `wUpDown_t` object on the heap and return a pointer to that object in ESI. If you make a standard object call to this constructor, then `create_wUpDown` will initialize that object in-place.

wudName: HOWL assigns this string to the _name data field of the object. This string's value should be constant over the execution lifetime of the newly initialized object.

parent: this is the handle of the `wView_t` or `wForm_t` object on which the track bar will be drawn.

alignment: this is one of the following Up/Down window styles:

`UDS_ALIGNLEFT`	Positions the up-down control next to the left edge of the buddy window. The buddy window is moved to the right and its width decreased to accommodate the width of the up-down control.
`UDS_ALIGNRIGHT`	Positions the up-down control next to the right edge of the buddy window. The width of the buddy window is decreased to accommodate the width of the up-down control.
`UDS_ARROWKEYS`	Causes the up-down control to increment and decrement the position when the UP ARROW and DOWN ARROW keys are pressed.
`UDS_AUTOBUDDY`	Automatically selects the previous window in the Z order as the up-down control's buddy window.

UDS_HORZ	Causes the up-down control's arrows to point left and right instead of up and down.
UDS_NOTHOUSANDS	Does not insert a thousands separator between every three decimal digits.
UDS_SETBUDDYINT	Causes the up-down control to set the text of the buddy window (using the WM_SETTEXT message) when the position changes. The text consists of the position formatted as a decimal or hexadecimal string.
UDS_WRAP	Causes the position to "wrap" if it is incremented or decremented beyond the ending or beginning of the range.

x, y, width, height: These arguments form a bounding box in which the up/down arrow will be drawn. If `buddy` contains a non-NULL value, then Windows will ignore these values and will, instead, attach the arrows to the buddy edit box.

lowerRange: This is the minimum value that the up/down arrow control will decrement to. This value is typically zero, but it can be any 16-bit value that is less than the `upperRange` value. If the up/down control's current value is equal to `lowerRange` and the user presses on the down arrow, Windows will ignore the decrement request.

upperRange: This is the maximum value that the up/down arrow control will increment to. This value can be any 16-bit value that is greater than the `lowerRange` value. If the up/down control's current value is equal to `upperRange` and the user presses on the up arrow, Windows will ignore the increment request.

initialPosn: This is the initial value of the up/down arrow control. It must be a 16-bit value in the range `lowerRange..upperRange`.

onClick: this is the address (which can be NULL) of an `onClick` widgetProc procedure. The constructor will initialize the `onClick` field with this value. (Note that the `onClick` field is inherited from the `wClickable_t` parent class).

get_lowerRange	This method retrieves the 16-bit lower range value for the up/down arrow control. It returns this 16-bit value (zero extended) in the EAX register.
get_upperRange	This method retrieves the 16-bit upper range value for the up/down arrow control. It returns this 16-bit value (zero extended) in the EAX register.
get_position	This method retrieves the 16-bit position value for the up/down arrow control. It returns this 16-bit value (zero extended) in the EAX register.
set_lowerRange	This method sets the 16-bit lower range value for the up/down arrow control. Applications should limit the parameter to a 16-bit value.
set_upperRange	This method sets the 16-bit upper range value for the up/down arrow control. Applications should limit the parameter to a 16-bit value.
set_position	This method sets the 16-bit position value for the up/down arrow control. Applications should limit the parameter to a 16-bit value.
processMessage	This is a private method. Applications must not call this method.

38.3.18.2 wUpDownEditBox_t

The `wUpDownEditBox_t` class is used to create a combination edit box and up/down arrow on a form.

```
wUpDownEditBox_t:
    class inherits( wabsEditBox_t );
        var
            wUpDownEditBox_private:
                record
```

```
                    lowerRange      :dword;
                    upperRange      :dword;
                    upDownHandle    :dword;
                    upDownStyle     :dword;
                    onUpDown        :widgetProc;

                endrecord;

            procedure create_wUpDownEditBox
            (
                wudName     :string;
                initialTxt  :string;
                parent      :dword;
                style       :dword;
                upDownStyle :dword;
                x           :dword;
                y           :dword;
                width       :dword;
                height      :dword;
                lowerRange  :dword;
                upperRange  :dword;
                initialPosn :dword;
                onChange    :widgetProc;
                onUpDown    :widgetProc
            );  external;

            method get_lowerRange;  @returns( "eax" );          external;
            method get_upperRange;  @returns( "eax" );          external;
            method get_position;    @returns( "eax" );          external;

            method set_lowerRange( lowerRange:word );           external;
            method set_upperRange( upperRange:word );           external;
            method set_position(   position  :word );           external;

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

            override method processMessage;                     external;

        endclass;
```

`create_wUpDownEditBox`

This is the consructor for the `wUpDownEditBox_t` class. If you call this as a class procedure (e.g., "wUpDownEditBox_t.create_wUpDown") then this procedure will allocate storage for a new `wUpDownEditBox_t` object on the heap and return a pointer to that object in ESI. If you make a standard object call to this constructor, then `create_wUpDown` will initialize that object in-place.

wudName: HOWL assigns this string to the _name data field of the object. This string's value should be constant over the execution lifetime of the newly initialized object.

parent: this is the handle of the `wView_t` or `wForm_t` object on which the track bar will be drawn.

alignment: this is one of the following Up/Down window styles:

`UDS_ALIGNLEFT` Positions the up-down control next to the left edge of the buddy window. The buddy window is moved to

the right and its width decreased to accommodate the width of the up-down control.

`UDS_ALIGNRIGHT` Positions the up-down control next to the right edge of the buddy window. The width of the buddy window is decreased to accommodate the width of the up-down control.

`UDS_ARROWKEYS` Causes the up-down control to increment and decrement the position when the UP ARROW and DOWN ARROW keys are pressed.

`UDS_AUTOBUDDY` Automatically selects the previous window in the Z order as the up-down control's buddy window.

`UDS_HORZ` Causes the up-down control's arrows to point left and right instead of up and down.

`UDS_NOTHOUSANDS` Does not insert a thousands separator between every three decimal digits.

`UDS_SETBUDDYINT` Causes the up-down control to set the text of the buddy window (using the WM_SETTEXT message) when the position changes. The text consists of the position formatted as a decimal or hexadecimal string.

`UDS_WRAP` Causes the position to "wrap" if it is incremented or decremented beyond the ending or beginning of the range.

x, y, width, height: These arguments form a bounding box in which the up/down arrow will be drawn. If `buddy` contains a non-NULL value, then Windows will ignore these values and will, instead, attach the arrows to the buddy edit box.

lowerRange: This is the minimum value that the up/down arrow control will decrement to. This value is typically zero, but it can be any 16-bit value that is less than the `upperRange` value. If the up/down control's current value is equal to `lowerRange` and the user presses on the down arrow, Windows will ignore the decrement request.

upperRange: This is the maximum value that the up/down arrow control will increment to. This value can be any 16-bit value that is greater than the `lowerRange` value. If the up/down control's current value is equal to `upperRange` and the user presses on the up arrow, Windows will ignore the increment request.

initialPosn: This is the initial value of the up/down arrow control. It must be a 16-bit value in the range `lowerRange..upperRange`.

onClick: this is the address (which can be NULL) of an `onClick` widgetProc procedure. The constructor will initialize the `onClick` field with this value. (Note that the `onClick` field is inherited from the `wClickable_t` parent class).

`get_lowerRange` This method retrieves the 16-bit lower range value for the up/down arrow control. It returns this 16-bit value (zero extended) in the EAX register.

`get_upperRange` This method retrieves the 16-bit upper range value for the up/down arrow control. It returns this 16-bit value (zero extended) in the EAX register.

`get_position` This method retrieves the 16-bit position value for the up/down arrow control. It returns this 16-bit value (zero extended) in the EAX register.

`set_lowerRange` This method sets the 16-bit lower range value for the up/down arrow control. Applications should limit the parameter to a 16-bit value.

`set_upperRange` This method sets the 16-bit upper range value for the up/down arrow control. Applications should limit the parameter to a 16-bit value.

`set_position` This method sets the 16-bit position value for the up/down arrow control. Applications should limit the parameter to a 16-bit value.

processMessage This is a private method. Applications must not call this method.

38.3.19 Icons

Icons are generally 16x16 or 32x32 bitmapped objects drawn on the screen.

38.3.19.1 wIcon_t

A wIcon_t object on a form can draw a user-defined icon or a system icon.

```
wIcon_t:
    class inherits( wSurface_t );

        var
            align( 4 );
            wIcon_private:
                record

                    iconName    :string;
                    iconHandle  :dword;

                endrecord;

            procedure create_wIcon
            (
                wiName      :string;
                iconName    :string;
                parent      :dword;
                x           :dword;
                y           :dword;
                width       :dword;
                height      :dword;
                bkgColor    :dword
            );  external;

            method get_iconName;    @returns( "eax" );      external;
            method load_icon( iconName:string );            external;
            override method destroy;                        external;
            override method processMessage;                 external;

    endclass;
```

`iconName` This is either the name of an icon resource within the executable file, or a special numeric value (less than $1_0000) that specifies a system icon. Note that this field does not contain the name of an icon file on the disk. Applications should not set the value of this field directly. Instead, they should use the constructor or the `load_icon` method to set the icon name.

`iconHandle` This is a handle for the icon resource associated with thie `wIcon_t` object. This is a private field and applications should not read or write its value.

`create_wIcon` This is the consructor for the `wIcon_t` class. If you call this as a class procedure (e.g., "wIcon_t.create_wIcon") then this procedure will allocate storage for a new `wIcon_t` object on the heap and return a pointer to that object in ESI. If you make a standard object call to this constructor, then `wIcon` will initialize that object in-place.

wiName: HOWL assigns this string to the _name data field of the object. This string's value should be constant over the execution lifetime of the newly initialized object.

iconName: this is the name of the icon resource in the executable file to associate with the icon (if this value is a string), or it is a value less than $1_0000 that specifies a system icon value. The following are valid non-string values to supply to this parameter: w.IDI_APPLICATION, w.IDI_ASTERISK, w.IDI_EXCLAMATION, w.IDI_HAND, w.IDI_QUESTION, and w.IDI_WINLOGO.

parent: this is the handle of the `wView_t` or `wForm_t` object on which the icon will be drawn.

x, y, width, height: These arguments form a bounding box in which the up/down arrow will be drawn. If `buddy` contains a non-NULL value, then Windows will ignore these values and will, instead, attach the arrows to the buddy edit box.

bkgColor: this is the RGB background color that HOWL will fill the bounding rectangle with if the bounding rectangle is larger than the icon.

`load_icon` This method loads the icon resource whose name you specify as the string parameter into the icon object. The parameter must either be the (string) name of an icon resource in the executable file or one of the system-defined icon constants (see the discussion for the constructor).

`destroy` This is the destructor for the wIcon_t object. Applications won't normally call this destructor for icons attached to some form (or other container) as destroying that container will automatically invoke the destructor for the icon. However, if you create a stand-alone icon object and don't attach it to some container object, then you should call the icon's destructor when you are done with the icon to free the associated system resources.

`processMessage` This is a private method. Applications must not call it.

38.3.20 Text

HOWL provides two classes for dealing with text: the `wFont_t` class and the `wLabel_t` class. Unlike most concrete classes in HOWL, `wFont_t` objects are not widgets (that is, controls that appear visually on a form). Font objects are associated with `wLabel_t` objects (which do appear on a form) and other text, but are not visual items themselves.

38.3.20.1 wFont_t

Font objects represent a particular typeface for use by `wLabel_t` and other text objects in HOWL. Note that `wFont_t` objects are somewhat unique amongst the HOWL concrete classes insofar as there is no statement in the HOWL declarative language to create a font object. In general, an application will create all the fonts it needs in the `appStart` procedure and it will call the destructors for those font objects in the `appTerminate` procedure.

```
wFont_t:
    class inherits( wBase_t );

        var
            align( 4 );
            wFont_private:
                record

                    family          :byte;
                    bold            :boolean;
                    italic          :boolean;
                    underline       :boolean;
                    strikeout       :boolean;
                    monospaced      :boolean;
                    align( 4 );

                    faceName        :string;
                    size            :uns32;
```

```
            endrecord;

        procedure create_wFont
        (
            wfName          :string;
            parentHandle    :dword;
            faceName        :string;
            family          :byte;
            size            :uns32;
            bold            :boolean;
            italic          :boolean;
            underline       :boolean;
            strikeout       :boolean;
            monospaced      :boolean
        );  external;

        override method destroy;                    external;

        // Accessor functions:

        method get_facename;    @returns( "eax" );  external;
        method get_size;        @returns( "eax" );  external;
        method get_family;      @returns( "al" );   external;
        method get_bold;        @returns( "al" );   external;
        method get_italic;      @returns( "al" );   external;
        method get_underline;   @returns( "al" );   external;
        method get_strikeout;   @returns( "al" );   external;
        method get_monospaced;  @returns( "al" );   external;


    endclass;
```

`faceName` This string is the name of the Windows typeface to use for the font. This is a string such as "Courier New" or "Times New Roman". If this field is NULL or is the empty string, then Windows will pick an appropriate font that matches the other font characteristics. If Windows cannot find the specified font name, it will find the closest one that matches the font characteristics you provide. Applications should not write values to this field.

`size` This is the size, in points, for the typeface. Applications should not write values to this field.

`family` This is one of the following constants:

w.FF_DECORATIVE	Novelty fonts. Old English is an example.
w.FF_DONTCARE	Don't care or don't know.
w.FF_MODERN	Fonts with constant stroke width, with or without serifs. Pica, Elite, and Courier New® are examples.
w.FF_ROMAN	Fonts with variable stroke width and with serifs. MS® Serif is an example.
w.FF_SCRIPT	Fonts designed to look like handwriting. Script and Cursive are examples.
w.FF_SWISS	Fonts with variable stroke width and without serifs. MS Sans Serif is an example.

Applications should not write values to the `family` field.

`bold` True for boldfaced fonts, false for normal weight fonts. Applications should not write values to this field.

`italic`	True for italic slant fonts, false for normal fonts. Applications should not write values to this field.
`underline`	True for underlined fonts, false for normal fonts. Applications should not write values to this field.
`strikeout`	True for strikeout fonts, false for normal fonts. Applications should not write values to this field.
`monospaced`	True for monospaced fonts, false for proportional fonts. Applications should not write values to this field.

create_wFont	This is the consructor for the `wFont_t` class. If you call this as a class procedure (e.g., "wFont_t.create_wFont") then this procedure will allocate storage for a new `wFont_t` object on the heap and return a pointer to that object in ESI. If you make a standard object call to this constructor, then `create_wFont` will initialize that object in-place. **wfName**: HOWL assigns this string to the _name data field of the object. This string's value should be constant over the execution lifetime of the newly initialized object. The remaining parameters all correspond to the data fields for this class. See their descriptions for more details. Calling the constructor is the only legal way to write values to the `wFont_t` class' data fields.
`destroy`	This is the destructor for the wFont_t class. Because fonts aren't normally attached to a form (and you cannot create them via the HOWL declarative language), it is usually the applications responsibility to call the destructor for a font when the application is done using it. Alternately, an application can insert a font into a container object and let that container destroy the font object when the container is destroyed.

38.3.20.2 wLabel_t

A `wLabel_t` object displays static text on a form. Actually, "static" is a bit of a misnomer because an application can change the text will a method call, but Windows, HOW, and the user do not change this text behind the application's back.

```
wLabel_t:
    class inherits( wVisual_t );
        var
            align( 4 );
            wLabel_private:
                record

                    caption     :string;
                    font        :wFont_p;
                    alignment   :dword;
                    foreColor   :dword;

                endrecord;

        procedure create_wLabel
        (
            wlName      :string;
            caption     :string;
            parent      :dword;
            x           :dword;
            y           :dword;
            width       :dword;
            height      :dword;
            alignment   :dword;
            foreColor   :dword;
```

```
            bkgColor    :dword
        );  external;


        override method destroy;                            external;
        override method processMessage;                     external;

        method get_font;        @returns( "eax" );          external;
        method get_caption;     @returns( "eax" );          external;
        method a_get_caption;   @returns( "eax" );          external;
        method get_foreColor;   @returns( "eax" );          external;

        method set_font( font:wFont_p );                    external;
        method set_caption( caption:string );               external;
        method set_foreColor( foreColor:dword );            external;

    endclass;
```

`caption` This data field points at the textual data that the wLabel_t object will display on a form. Applications should never read or write this data field directly, they should always use the get_caption and set_caption accessor/mutator methods to manipulate this string.

`font` This is a pointer to a `wFont_t` object or the value NULL. If this field contains NULL, Windows will pick an appropriate system font and use that to draw the label's text. If this field contains a pointer to a `wFont_t` object, this Windows will use that particular font to draw the label's text. Applications should never read or write this data field directly, they should always use the get_font and set_font accessor/mutator methods to manipulate this value.

`alignment` Applications should not write to this field. It's value is set by the constructor. This field should contain one of the following constants:

w.DT_BOTTOM	Bottom-justifies text. This value must be combined with DT_SINGLELINE.
w.DT_CENTER	Centers text horizontally.
w.DT_EXPANDTABS	Expands tab characters. The default number of characters per tab is eight.
w.DT_LEFT	Aligns text to the left.
w.DT_NOPREFIX	Turns off processing of prefix characters. Normally, DrawText interprets the mnemonic-prefix character & as a directive to underscore the character that follows, and the mnemonic-prefix characters && as a directive to print a single &. By specifying DT_NOPREFIX, this processing is turned off.
w.DT_RIGHT	Aligns text to the right.
w.DT_SINGLELINE	Displays text on a single line only. Carriage returns and linefeeds do not break the line.
w.DT_TOP	Top-justifies text (single line only).
w.DT_VCENTER	Centers text vertically (single line only).
w.DT_WORDBREAK	Breaks words. Lines are automatically broken between words if a word would extend past the edge of the rectangle specified by the lpRect parameter. A carriage return-linefeed sequence also breaks the line.

`foreColor` This is the foreground color for the text (the RGB color used to draw the actual characters). Applications should never read or write this data field directly, they should always use the get_foreColor and set_foreColor accessor/mutator methods to manipulate this value.

create_wLabel	This is the consructor for the wLabel_t class. If you call this as a class procedure (e.g., "wLabel_t.create_wLabel") then this procedure will allocate storage for a new wLabel_t object on the heap and return a pointer to that object in ESI. If you make a standard object call to this constructor, then create_wLabel will initialize that object in-place. **wlName**: HOWL assigns this string to the _name data field of the object. This string's value should be constant over the execution lifetime of the newly initialized object. **caption**: this is string data HOWL will associate with the label object. Note that HOWL will make an internal copy (on the heap) of this string. **parent**: this is the handle of the wView_t or wForm_t object on which the label will be drawn. **x, y, width, height**: These arguments form a bounding box in which the up/down arrow will be drawn. If buddy contains a non-NULL value, then Windows will ignore these values and will, instead, attach the arrows to the buddy edit box. **alignment**: this is one of the alignment constants described under the alignment field, earlier. It this field is zero, w.DT_LEFT is the default value. **foreColor**: this is an RGB value that specifies the foreground color for the label. **bkgColor**: this is an RGB value that specifies the background color for the label.
destroy	This is the destructor for the label class. Normally, the container form holding the label will call this destructor automatically when you destroy the form. If you create a label manually and don't insert it into the widget list of a container object, then you will need to call this destructor yourself.
processMessage	This is a private method. Applications should not call this method.
get_font	This is the accessor method that returns a pointer to the wFont_t object (or NULL) pointed at by the font data field.
get_caption	This returns the caption string pointer in EAX for read-only access. The application must not modify this string or HOWL will display inconsistent results.
a_get_caption	This method returns a pointer to a copy of the caption string (that it allocates on the heap) in the EAX register. It is the caller's responsibility to free the storage associated with this string when it is done using the string data.
get_foreColor	This accessor method returns the current foreground color that the wLabel_t object uses to draw the text.
set_font	This method is the mutator function for the font data field. Applications should always call this method to change the font of a wLabel_t object.
set_caption	This method is the mutator function for the caption data field. Applications should always call this method to change the string associated with a wLabel_t object.
set_foreColor	This method is the mutator function for the foreColor data field. Applications should always call this method to change the foreground color associated with a wLabel_t object.

38.3.21 Views, Windows, and Tab Pages

A view is a container object that is also a surface upon which you can draw or place other widgets. In this sense, a view is very similar to a wForm_t object. Actually, a view is equivalent to the client area of a wForm_t object; that is, it's the application window without the title bar or system menu components.

38.3.21.1 wTabPage_t

The wTabPage_t class is basically a concrete implementation of the abstract window_t class.

```
wTabPage_t:
    class inherits( window_t );

        var
            align( 4 );
            wTabPage_private:
                record

                    pageHandler :widgetProc;

                endrecord;

        procedure create_wTabPage
        (
            wpName              :string;
            parentWindowHandle  :dword;
            handler             :widgetProc;
            x                   :dword;
            y                   :dword;
            width               :dword;
            height              :dword;
            fillColor           :dword
        );  external;

        override method processMessage;                 external;

    endclass;
```

`pageHandler` This is a `widgetProc` pointer to a user-defined `processMessage` function for the `wTabPage_t` object. If this pointer contains NULL, then HOWL will invoke the parent (`window_t`) processMessage handler. If all a `wForm_t` object is doing is containing other widgets, this is all that is necessary. However, if you're going to draw on the `wTabPage_t` object, then you will have to write your own `processMessage` function and intercept `w.WM_PAINT` and other messages. A discussion of the actual pageHandler procedure appears a little later in this section.

`create_wTabPage` This is the consructor for the `wTabPage_t` class. If you call this as a class procedure (e.g., "wTabPage_t.create_wTabPage") then this procedure will allocate storage for a new `wTabPage_t` object on the heap and return a pointer to that object in ESI. If you make a standard object call to this constructor, then create_`wTabPage` will initialize that object in-place.

wpName: HOWL assigns this string to the _name data field of the object. This string's value should be constant over the execution lifetime of the newly initialized object.

parentWindowHandle: this is the handle of the `wTabPage_t` or `wForm_t` object on which the `wTabPage_t` object will be placed.

x, y, width, height: These arguments form a bounding box in which the `wTabPage_t` object will be drawn.

fillColor: this is an RGB value that specifies the background color for the window.

`processMessage` This is a private method. Applications must not call it.

The `pageHandler` `widgetProc` needs some additional explanation. To begin with, although this pointer type is `widgetProc`, technically it ought to have the same prototype as the `processMessage` method (`hwnd`, `uMsg`, `wParam`, and `lParam` parameters). To overcome the difference in prototypes, HOWL passes to the `pageHandler` procedure the `uMsg` parameter value in the EAX register. The `hwnd` (window handle) argument is easily accessible as the handle field of the `wTabPage_t` object (accessible via the `thisPtr` parameter).

The pageHandler procedure is basically the Windows' wndproc procedure that handles the messages for the wTabPage_t object (that is, it's equivalent to a `wTabPage_t.processMessage` function). If you want to draw on the `wTabPage_t surface`, you'll need to intercept Windows' messages such as `w.WM_PAINT` and so on. Most of the time, you'll only want to handle a few of the messages sent to a `wTabPage_t` object yourself and you'll want to pass the rest of them on to a default message handler. In a pure Win32 application, you'd accomplish this by calling the `w.DefWindowProc` procedure. In an object-oriented system such as HOWL, however, what you really want to do is call the parent class' message handler (that is, `window_t.processMessage` in the case of a `wTabPage_t` object). Unfortunately, it's difficult to do this directly, so HOWL cheats and defines a special class procedure in the `window_t` class that you can call directly that is an alias for the `window_t.processMessage` method. This class procedure is called `window_t.view_t_processMessage` (so there is no mistaking its purpose). If ESI contains a pointer to the wTabPage_t object (and it must when you call `window_t.view_t_processMessage`), then you can call this procedure with the following statement:

```
(type window_t [esi]).view_t_processMessage
```

Here is a sample `pageHandler` procedure that does exactly the same thing as would happen if the `pageHandler` pointer were NULL (which is to call the `window_t.processMessage` method):

```
simplePH:widgetProc;
begin simplePH;

    mov( thisPtr, esi );  // Technically not needed, ESI contains this already
    (type window_t [esi]).view_t_processMessage  // Call special alias proc
    (
        (type wTabPage_t [esi]).handle, // Handle to the wTabPage_t window
        eax,                        // Message code passed to us in EAX
        wParam,                     // Pass on the wParam value
        lParam                      // Pass on the lParam value
    );

end simplePH;
```

Of course, it doesn't make much sense to do nothing more than call the view_t_processMessage procedure. You could leave the `pageHandler` NULL and the wTabPage_t class would automatically do this for you (more efficiently, too). In general, you'll check for a few messages you need to handle and call view_t_processMessage as the default handler, e.g.,:

```
typicalPH:widgetProc;
begin typicalPH;

  if( eax = someMessage ) then

    // code to handle someMessage

  elseif( eax = someOtherMessage ) then

    // code to handle someOtherMessage

  else  // Default case

    mov( thisPtr, esi );  // Technically not needed, ESI contains this already
    (type window_t [esi]).view_t_processMessage  // Call special alias proc
    (
        (type wTabPage_t [esi]).handle, // Handle to the wTabPage_t window
        eax,                        // Message code passed to us in EAX
        wParam,                     // Pass on the wParam value
        lParam                      // Pass on the lParam value
    );

  endif;
```

```
end typicalPH;
```

Please see the appropriate Windows documentation for details on how to handle message in a wndproc (message handling) procedure.

38.3.21.2 wView_t

`wView_t` objects are a concrete implementation of the `wSurface_t` class. You use `wView_t` objects to create bordless windows on which to draw things. Note that `wView_t` is not a container class (like `window_t`), so you cannot place other objects on a `wView_t` window. Please see the discussion of the `wSurface_t` class for more details concerning the capabilities of a `wView_t` object.

```
wView_t:
    class inherits( wSurface_t );

        procedure create_wView
        (
            wsName      :string;
            exStyle     :dword;
            style       :dword;
            parent      :dword;
            x           :dword;
            y           :dword;
            width       :dword;
            height      :dword;
            bkgColor    :dword;
            visible     :boolean
        );  external;

    endclass;
```

38.3.21.3 window_t

The `window_t` type is functionally equivalent to `wSurface_t` with on major difference: the `window_t` class is derived from `wContainer_t` rather than directly from `wVisual_t`, so `window_t` objects can contain other widgets.

```
window_t:
    class inherits( wContainer_t );

        var
            align( 4 );
            window_private:
                record

                    // onPaint event pointer:

                    onPaint     :widgetProc;

                endrecord;


        procedure create_window
        (
            wwName      :string;
            caption     :string;
            exStyle     :dword;
            style       :dword;
            parent      :dword;
```

```
                x           :dword;
                y           :dword;
                width       :dword;
                height      :dword;
                bkgColor    :dword;
                visible     :boolean
            );  external;

            override method destroy;                        external;
            override method processMessage;                 external;
            override method onClose;                        external;
            override method onCreate;                       external;

            method get_onPaint;     @returns( "eax" );      external;
            method set_onPaint( onPaint:widgetProc );       external;

            procedure view_t_processMessage
            (
                hwnd    :dword;
                uMsg    :dword;
                wParam  :dword;
                lParam  :dword
            );  external( "window_t_processMessage" );

        endclass;
```

`onPaint` This widgetProc pointer is either NULL, or it points at a widgetProc procedure that HOWL will call whenever Windows sends the form a w.WM_PAINT message.

`destroy` This method is the destructor for the `window_t` class. Because the `window_t` class is derived from `wContainer_t`, invoking this destructor will also (recursively) invoke the destructors of all the widgets held by the `window_t` object. This destructor will also free up the system brush resource (held by `_bkgBrush`) and free any storage associated with the `window_t` object. As is true for all abstract base classes, an application will not directly call this destructor method. Instead, the application will call the destructor of a derived class which, in turn, will call this destructor.

`processMessage`,

`onClose`,

`onCreate` These are private methods in the class. Applications should not call them.

`get_onPaint`

`set_onPaint` These are the accessor/mutator methods for the `onPaint` field. Applications must use these methods to access or modify the value of the `onPaint` field.

`view_t_processMessage`

Most of the time the rule is that applications do not call the `processMessage` method. In the case of the wTabPage_t derived class, however, there is a special case where an application needs to call the `window_t.processMessage` method. Normally, this is a somewhat difficult thing to do (especially from a `widgetProc`, from where the call is going to be made). The `view_t_processMessage` class procedure is actually an alias for the `window_t.processMessage` method, so that a `wTabPage_t` object's `pageHandler` can call `window_t.processMessage` in that special case. In general, applications should never call `view_t_processMessage` except in this one special case. Please see the discussion of `wTabPage_t` for more details.

38.3.22 Timers

HOWL provides a wTimer_t class that lets you create "one-shot" or "periodic" timers in your HOWL applications.

38.3.22.1 wTimer_t

`wTimer_t` objects are non-visual objects that can automatically call a `widgetProc` for you whenever a certain amount of time (in milliseconds) elapses. `wTimer_t` objects are useful for passing control to your code on a periodic basis even if there are no user interface interactions with your application's form.

Timers are unique amongst the widgets. You will notice that the `wTimer_t` class is derived from the `wVisual_t` class. However, `wTimer_t` objects aren't exactly visual because such objects don't appear visually on a form. `wTimer_t` objects need to be actual `window_t` objects so that they can have a Windows' `wndProc` message handler that can receive messages from a timing thread and appear in a widget list of some container object (the window itself is just a 1x1 pixel window at position (0,0) on your form). Normally, this window is hidden from the user. You should take care not to call the `show` method to make it visible.

`wTimer_t` objects operate in one of two modes: periodic and one-shot. If you initialize a `wTimer_t` object with the constant `wTimer_t.oneShot` and then start the timer, it will run for the specified amount of time (in milliseconds) and then call an `onTimeOut widgetProc` exactly once. This is useful if you need exactly one notification at some future time.

`wTimer_t` objects can also operate in periodic mode. If you initialize the `wTimer_t` object with the `wTimer_t.periodic` constant, then it will automatically call an `onTimeOut widgetProc` (approximately) every `period` milliseconds until you explicitly stop the timer.

Note that although a `wTimer_t` object spawns a thread to handling the timing chores, that thread does not call any HLA Standard Library code, and invoking the `onTimeOut widgetProc` is done via a `w.PostMessage` Win32 call, so you don't have to compile the program with the HLA "-thread" command-line parameter and you don't have to worry about multi-threaded synchronization. The `onTimeOut` procedure always executes in the same thread as your main program.

```
wTimer_t:
    class inherits( wVisual_t );

        const
            oneShot     := 0;
            periodic    := 1;

        static
            messageCode     :dword := 0;

        var
            align( 4 );
            wTimer_private:
                record

                    // Timeout value in milliseconds:

                    period          :dword;

                    // oneShot or periodic

                    timing          :dword;

                    // Widget proc to call on time out:

                    onTimeOut       :widgetProc;

                    // 1 = run, 0 = wait

                    trigger     :dword;

                    // Win32 thread handle for timer
```

```
                        threadHandle    :dword;

                endrecord;


            procedure create_wTimer
            (
                timerName       :string;
                parentHandle    :dword;
                periodInMsec    :dword;
                timing          :dword;
                onTimeOut       :widgetProc
            );  external;

            override method destroy;                                    external;
            override method processMessage;                             external;

            method  start;                                              external;
            method  stop;                                               external;

            method  get_onTimeOut;  @returns( "eax" );                  external;
            method  set_onTimeOut( onTimeOut:widgetProc );              external;
            method  get_period;     @returns( "eax" );                  external;
            method  set_period( period:dword );                         external;
            method  get_timing;     @returns( "eax" );                  external;
            method  set_timing( timing:dword );                         external;

            // Apps must never call this, it is put here for
            // convenience (to be able to use "this"):

            procedure _timerThread( wTimerObj:wTimer_p );               external;


        endclass;
```

`wTimer_t.oneShot` This is the constant you specify as the timing argument to `create_wTimer` or `set_timing` if you want to create a one-shot timer object.

`wTimer_t.periodic`

This is the constant you specify as the timing argument to `create_wTimer` or `set_timing` if you want to create a free-running `periodic` timer object.

`period` This is the time, in milliseconds, that a `wTimer_t` object will delay before posting a message that will call the `onTimeOut widgetProc`. This is a private data field. Applications should not access it directly but should, instead, use the provided accessor/mutator methods.

`timing` This field determines the type of the timer. It will contain one of the following constants:

wTimer_t.oneShot: the timer, when started, will call the `onTimeOut widgetProc` exactly once and then disable itself.

wTimer_t.periodic: the timer, when started, will call the `onTimeOut widgetProc` about every `period` milliseconds until you explicitly stop the timer.

This is a private data field. Applications should not access it directly but should, instead, use the provided accessor/mutator methods.

`onTimeOut` This is the address of the widgetProc that HOWL will call when the timer times out. If this field is NULL, HOWL will not call any widgetProc and the timeout will go unnoticed by the application. This is a private data field. Applications should not access it directly but should, instead, use the provided accessor/mutator methods.

`_trigger`	This is an internal field that the `wTimer_t` object uses to communicate between the main thread and the timer thread. Applications should never access this field.
`_threadHandle`	This is an internal field that the `wTimer_t` object uses to communicate between the main thread and the timer thread. Applications should never access this field.
`create_wTimer`	This is the class constructor for `wTimer_t` objects. If you call this as a class procedure (e.g., "wTimer_t.create_wTimer") then this procedure will allocate storage for a new `wTimer_t` object on the heap and return a pointer to that object in ESI. If you make a standard object call to this constructor, then `create_wTimer` will initialize that object in-place. Note that calling the constructor does not start the timer running. You must explicitly call the `wTimer_t` object's start method to start the timer running. This procedure has the following parameters: **timerName**: a string specifying the timer variable's name (in HLA). **parentHandle**: the handle of the `wForm` object on which the timer is attached. **periodInMsec**: this is the initial timeout period for the timer. The constructor will initialize the period field with this value. You can use the accessor/mutator methods to change this value later. **timing**: this is the initial type of the timer. This value should either be `wTimer_t.oneShot` or `wTimer_t.periodic`. You can use the accessor and mutator methods to change the timer type at a later time. **onTimeOut**: this is either NULL or the address of a `widgetProc` that the timer will call whenever the timer times out. You can use the accessor and mutator methods to change the timeout `widgetProc` at a later time.
`destroy`	This is the destructor for the `wTimer_t` object. Normally, `wTimer_t` objects are inserted onto a widget list and the container automatically calls the destructor when the container is destroyed. However, if you create a `wTimer_t` object and you don't attach it to some container's widget list, then you will need to explicitly call the destructor to free up the timer object.
`processMessage`	This is an internal procedure. Applications must not call it.
`get_onTimeOut`, `get_period`, `get_timing`	These are the accessor methods for the corresponding data fields in the class. Applications should call these methods to access the data fields rather than reading their values directly.
`set_onTimeOut`, `set_period`, `set_timing`	These are the mutator methods for the corresponding data fields in the class. Applications should call these methods to access the data fields rather than writing their values directly. Note that you should take care when setting these values while a timer is actually running. Although access to these objects is synchronized (you don't have to worry about thread problems), changing these values while a timer is operating can make your programs difficult to read and modify and it can introduce some obscure bugs.
`start`	This starts the timer. If the timer was already running, this will kill the current timer thread and start a new one. When the timer expires (after no sooner than `period` milliseconds), the timer thread will post a message to the `wTimer_t` object to tell it to call the `onTimeOut widgetProc`. Note that the `onTimeOut` procedure could actually be called much later, based on the number of messages in the Windows' message queue and the time it takes to process all those messages. Do not assume that exactly `period` milliseconds have passed since the call to `start` when your `onTimeOut` procedure begins execution. It could actually be much later than this.
`stop`	This method immediately stops the timer. This generally means that (if the timer is running) there will be no call to the `onTimeOut` procedure. However, the timer could have

already posted an `onTimeOut` call in the message queue, so you should not assume that there will be no call to `onTimeOut` after you call stop.