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

#6: Menus

In this sixth tutorial of this series, we'll take a look at implementing menu objects on HOWL forms.

Prerequisites:

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

Source Code:

The source code for the examples appearing in this tutorial are available as part of the HLA Examples download. You'll find the sample code in the Win32/HOWL subdirectories in the unpacked examples download. This particular tutorial uses the files *014_menu1.hla* and *014x_menu1.hla*.

Menus:

Pull-down menus are attached to the main form in a HOWL application. Because of the flexibility of Windows' menuing structure, menus are among the most complex items in the HOWL Declarative Language (HDL) and are positively gruesome if you decide to manually implement menus in your application.

The first thing to note about Windows' menu objects is that there are several different types of menu items you can work with. There is, of course, the main menu (which consists of several pull-down menu items along the menu bar, which is located just below the title bar on a form). Main menu items can be either menu separators, plain menu entries, checked menu entries, or submenu items.

Within a HOWL application, a programmer defines a main menu using the `wMainMenu..endwMainMenu` statement. The syntax for this sequence is very straight-forward:

```
wMainMenu;
```

```
      .
      .
      .
    << menu declaration statements >>
      .
      .
      .
endwMainMenu;
```

Within the main menu sequence, you can have `wSubMenu..endwSubMenu` and `wMenuItem` statements. Main main menu entries will be usually be submenus (created with the `wSubMenu..endwSubMenu` statement), though it is possible to create non-submenu menu entries directly on the main menu.

To attach a menu entry to the main menu, you use the `wMenuItem` HDL statement. This statement has the following syntax:

```
wMenuItem
(
    HLAid,            // HLA identifier used for menu item in main form class
    checkable,        // True if this menu item can have check marks, false if not
    "text",           // Text to display in the actual menu (bar)
    widgetProc        // WidgetProc to call if this menu item is selected
);
```

The `checkable` argument is the only non-obvious item here. In submenus, it is possible to create menu items that can have a check mark next to them. This field would contain true if you want to allow that check mark to appear, it would be false if you want to create a menu item that cannot have a check mark. Note that this argument should always contain false for main menu items.

As noted earlier, though it is possible to create menu entries on the main menu bar, generally the main menu bar only contains submenus (actual menu entries generally appear only in submenus). To create a submenu, you use the `wSubMenu..endwSubMenu` sequence. This statement uses the following syntax:

```
    wSubMenu( HLAid, "text" );
    .
    .
    .
  << submenu declaration statements >>
    .
    .
    .
    endwSubMenu;
```

The `HLAid` argument is a valid HLA identifier that HOWL inserts into the main form's class definition to hold a pointer to the submenu's object data. The "text" argument is the string you want to display for the submenu entry in the main (or sub) menu.

The submenu declaration statements may consist of `wMenuItem` entries, nested `wSubMenu..endwSubMenu` statements, and `wMenuSeparator` statements. Here is the syntax for a `wMenuSeparator` statement:

```
wMenuSeparator
```

A wMenuSeparator statement may only appear within a wSubMenu..endwSubMenu sequence (that is, it may not appear in a main menu). The wMenuSeparator statement causes Windows to draw a horizontal line across the menu to separate menu items in a submenu.

Here’s a sample menu declaration (from this chapter’s tutorial):

```
wForm( mainAppWindow );

    wMainMenu;

        wSubMenu( file_1, "file" );

            wMenuItem( dummy1, false, "dummy", nullProc );
            wMenuSeparator;
            wMenuItem( menu_exit, false, "Exit", onQuit );

        endwSubMenu;

    endwMainMenu;

endwForm
```

The HOWL wMenuItem_t and wMenu_t Classes

Before rushing in an looking at some example code, let’s pause for a moment and consider the two menu related classes in HOWL. First, let’s start with the wMenuItem_t class:

```
    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;
```

The private data fields have the following purposes:

`nextMenu`:	HOWL maintains a linked list of menu entries for the main form's menu. This field contains a pointer to the next `wMenuItem` entry in that linked list. Applications must never access this field.
`itemType`:	This field contains one of the following constants: `w.MF_STRING` for standard menu entries, `w.MF_CHECKED` for menu entries that support a check mark next to them, `w.MF_SEPARATOR` for a menu separator bar, or `w.MF_POPUP` for a submenu.
`itemString`:	This field specifies the text data that Windows will display for the menu item. This should be a relative short string.
itemHandler:	This field holds the address of the widgetProc that HOWL will call when the menu item is selected. Note that this field is only valid for `wMenuItem` entries that have the `w.MF_STRING` or `w.MF_CHECKED` item types.

The constructor for the `wMenuItem_t` class (`create_wMenuItem`) initializes (and possibly allocates storage for) a `wMenuItem_t` object. The constructor has the following arguments:

wmiName: This is the HLA identifier that the main form class has used for this menu item.
parentHandle: This is the handle of the main form (to which the menu is attached).
itemType: This is the value that will be used to initialize the `itemType` private data field.
itemString: This is the value that HOWL will use to initialize the `itemString` private data field.
itemHandler: this is the address of a widgetProc that HOWL will use to initialize the `item-Handler` private data field.

The `enable` and `disable` methods will enable or disable a particular menu item. If a menu item is disabled, it will be grayed and the user will not be able to select it from a menu.

The `get_*` and `set_*` methods are the accessor and mutator functions for `wMenuItem_t` private data fields.

The `wMenu_t` class defines a main menu item on a `wForm_t` object. Here is its definition:

```
wMenu_t:
    class inherits( wMenuItem_t );

        // Constructors/Destructors:

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



        override method destroy;                    external;

    endclass;
```

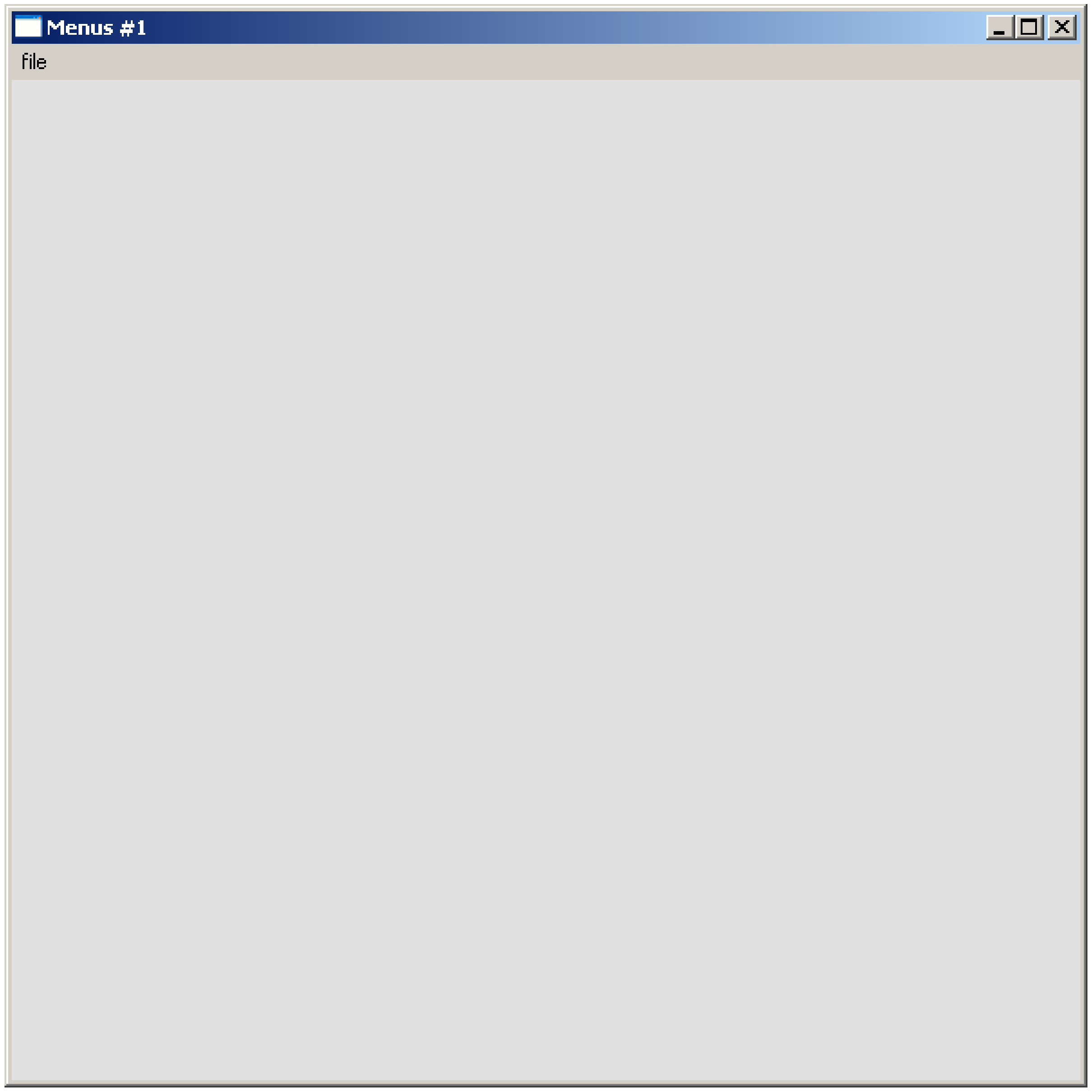
The constructor creates a new main menu item. The arguments have the following meaning:

wmName:	This string is an HLA identifier that should correspond to the name of the main menu item on the form. If you are using the HOWL Declarative Language (HDL) to define your menu items, the HDL will automatically create a main menu variable object named `*formname*_menu`. When manually creating a main menu (which is about the only reason you would call this constructor), you can use any name you want for the main menu object pointer but sticking to this convention is probably a good idea. Though the wmName string doesn't have to correspond to the actual variable name, it's probably a very good idea to do so. HOWL will initialize the `_name` field (from `wBase_t`) with a copy of this string argument's value.
wmText:	This would normally be the text that Windows displays for the menu entry. This is required because `wMenu_t` is a subclass of `wMenuItem_t` and it needs to initialize the `itemString` private data field. However, main menus don't actually have a string associated with them, so Windows ignores the value of this string. You should supply the empty string or a string like "main menu" for this argument. Do not supply NULL as the value of this argument because HOWL will make a copy of this string and NULL will cause the copy operation to fail.
parentHandle:	This must be the handle associated with the main form.

The `destroy` method is the destructor for the `wMenu_t` class. Generally an application will not directly call this destructor. The main form, when it gets destroyed, will automatically call this destructor for you.

The menu1 Application

With a discussion of the `wMenuItem_t` and `wMenu_t` classes out of the way, we can now get down to the business of looking at our sample program that uses the menu classes: *012_label.hla*. This sample program demonstrates placing a set of menus on a form. Here's what the screen looks like when you run this application.



Notice how this form displays a main menu just below the title bar. (Also note how the background color for the main form was changed slightly to make the menu bar stand out from the client area of the form.)

Here are the HDL statements that define this form:

```
wForm( mainAppWindow );

    wMainMenu;

        wSubMenu( file_1, "file" );

            wMenuItem( dummy1, false, "dummy", nullProc );
            wMenuSeparator;
            wMenuItem( menu_exit, false, "Exit", onQuit );

        endwSubMenu;

    endwMainMenu;



endwForm
```

The `dummy1` menu entry calls the `nullProc` widgetProc, which is the following:

```
proc nullProc:widgetProc;
begin nullProc;

    stdout.put( "Pressed the dummy  menu entry" nl );

end nullProc;
```

So when the users selects the “dummy” menu entry, the program will display this string to the console window.

The only other menu item is the `menu_exit` entry. It calls the `onQuit` widgetProc (the same one we’ve used for the quit button in all the previous tutorial examples) to terminate program execution.

Undeniably, this isn’t a very sophisticated menu example. The reason for its brevity is that I’m about to show you how much effort you save by using the HDL rather than manually coding the menu yourself. The wMenu..endwMenu, wSubMenu..endwSubMenu, wMenuItem, and wMenuSeparator statements generate a considerable amount of code and I wanted to keep the next example in this tutorial relatively short. For more sophisticated examples of menu declarations in HDL, you should check out the *howldemo.hla* and *howldemo.hhf* source files that are part of the HLA examples download. Another set of files to look at are the text editor tutorials appearing later in this tutorial series.

Here is the full source code for the *014_menu1.hla* application. One thing of interest to note here is that I’ve modified the `appStart` application to specify an RGB value for the form’s background color in order to differentiate the form’s client area from the menu bar.

```
// menu1-
//
//  This program demonstrates the use of menus on a form.

program menu1;
#linker( "comdlg32.lib" )
```

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

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

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

const
    applicationName := "Menus #1";
    formX           := w.CW_USEDEFAULT; // Let Windows position this guy
    formY           := w.CW_USEDEFAULT;
    formW           := 600;
    formH           := 600;




// Forward declarations for the onClick widgetProcs that we're going to
// call when an event occurs.

proc nullProc           :widgetProc; @forward;
proc onQuit             :widgetProc; @forward;



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

wForm( mainAppWindow );

    wMainMenu;

        wSubMenu( file_1, "file" );

            wMenuItem( dummy1, false, "dummy", nullProc );
            wMenuSeparator;
            wMenuItem( menu_exit, false, "Exit", onQuit );

        endwSubMenu;

    endwMainMenu;



endwForm


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

mainAppWindow_implementation();








// nullProc does nothing.
```

```
proc nullProc:widgetProc;
begin nullProc;

    stdout.put( "Pressed the dummy  menu entry" nl );

end nullProc;



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

proc onQuit:widgetProc;
begin onQuit;

    // Quit the app:

    w.PostQuitMessage( 0 );

end onQuit;





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

method mainAppWindow_t.onCreate;
begin onCreate;
end onCreate;



///////////////////////////////////////////////////////////////////////////////
//
//
// The following is mostly boilerplate code for all apps (about the only thing
// you would change is the size of the main app's form)
//
//
///////////////////////////////////////////////////////////////////////////////
//
// When the main application window closes, we need to terminate the
// application. This overridden method handles that situation.  Notice the
// override declaration for onClose in the wForm declaration given earlier.
// Without that, mainAppWindow_t would default to using the wVisual_t.onClose
// method (which does nothing).

method mainAppWindow_t.onClose;
begin onClose;

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

    w.PostQuitMessage( 0 );


end onClose;





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

procedure appStart;
begin appStart;

    push( esi );

    // Create the main application window:

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

end appStart;



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

procedure appTerminate;
begin appTerminate;

    // Clean up the main application's form.
    // Note that this will recursively clean up all the widgets on the form.
    mainAppWindow.destroy();

end appTerminate;


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

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

    raise( eax );

end appException;



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

begin menu1;

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

    // Use a special gray color to differentiate the menu bar
    // from the main client area on the form:

    mov( RGB(224, 224, 224), eax );
    mov( eax, howl.bkgColor_g );
    or( $FF00_0000, eax );
    mov( eax, howl.transparent_g );
    w.CreateSolidBrush( howl.bkgColor_g );
    mov( eax, howl.bkgBrush_g );

    // Start the HOWL Framework Main Program:

    HowlMainApp();

    // Delete the brush we created earlier:

    w.DeleteObject( howl.bkgBrush_g );

end menu1;
```

Manually Creating Menu Objects on a Form

For most objects you'll place on a form, using the HDL to define the form saves a fair amount of effort over manually coding those objects. When it comes to menus, however, using HDL saves you a *tremendous* amount of effort. In this second example in this chapter, I'm going to show you the manual implementation of the simplistic menu example from the previous section so that you can see how much effort using the HDL saves you when adding a menu to your forms.

First, let's begin with the class declaration we're going to need for the main form:

```
    mainAppWindow_t:
        class inherits( wForm_t );

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

            var

                mainAppWindow_menu      :wMenu_p;
                file_1                  :wMenu_p;
                dummy1                  :wMenuItem_p;
                menu_exit               :wMenuItem_p;
                 _wMenuSeparatorWidget_ :wMenuItem_p;
                align(4);


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

            override method onClose;
            override method onCreate;

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

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

        endclass;

        mainAppWindow_p :pointer to mainAppWindow_t;

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

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

The `mainAppWindow_menu` data field is the main menu object. The `file_1` data field is the pointer to the file submenu object that will appear in the main menu. The `dummy1` data field is the "dummy" menu entry that appears in the file menu. The `menu_exit` object corresponds to the exit menu entry that appears in the file menu. Lastly, the `_wMenuSeparatorWidget_` data field is used to create a menu separator item for the file menu. The remaining entries in the `mainAppWindow_t` class are the usual constructor and the `onCreate` and `onClose` methods (described in great detail in earlier tutorials).

The constructor for the `mainAppWindow_t` class is where we'll build the menu. Because menu items are appended to the main menu (this is just the way Windows works), we'll have to construct the menu in a "bottom-up" fashion -- constructing the lower-level menu entries first, then the submenus, and finally the main menu. Here's the constructor with all the steps annotated:

```
procedure mainAppWindow_t.create_mainAppWindow
(
        caption :string;
        exStyle :dword;
        style   :dword;
        parent  :dword;
        x       :dword;
        y       :dword;
        width   :dword;
        height  :dword;
        bkgClr  :dword;
        visible :boolean
);
var
    main    :mainAppWindow_p;
    rs      :wRadioSet_p;
    rsHndl  :dword;

begin create_mainAppWindow;
```

The beginning of the constructor is the same code from every other `mainAppWindow_t` constructor you'll write:

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

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

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

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

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

    // Initialize the VMT pointer:

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

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

    mov( cl, this.wBase_private.onHeap );

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

    mov( esi, main );
```

The first thing to do is to create the low-level menu items. We start with the “dummy” menu entry. The call to the `wMenuItem_t` constructor creates the menu item (allocating storage for it on the heap). This code then saves the object pointer in the dummy1 data field (in case the application wants to reference this object later on). Like all objects, we insert this menu item into the main form’s container list (so it will be automatically destroyed when the main form is destroyed) and then this code calls the main form’s `appendMenuItem` method to attach the “dummy” menu item to the main form’s menu. Note the use of the `w.MF_STRING` menu type (which is used for standard menu entries).

```
    // Code to create a wMenuItem object:

    mov(main,ebx);
    wMenuItem_t.create_wMenuItem("dummy1", ebx, w.MF_STRING, "dummy", &nullProc);
    mov( esi, (type mainAppWindow_t [ebx]).dummy1);
    mov(ebx,esi);
    this.insertWidget( this.dummy1);
    this.appendMenuItem( this.dummy1);
```

Because we have a menu separator in our simple menu application, we need to create a menu separator object. Here’s the code that does this; note the use of the `w.MF_SEPARATOR` type. Also note that the `itemString` and `itemHandler` values are irrelevant for menu separators, so we just supply NULL here (okay, even for the string, because HOWL won’t copy this data when creating a menu separator item):

```
    // Code to create a wMenuSeparator object:
```

```
mov(main,ebx);
wMenuItem_t.create_wMenuItem
(
    "_wMenuSeparatorWidget_",
    ebx,
    w.MF_SEPARATOR,
    NULL,
    NULL
);
mov( esi, (type mainAppWindow_t [ebx])._wMenuSeparatorWidget_);
main.insertWidget((type mainAppWindow_t [ebx])._wMenuSeparatorWidget_);
```

Note that the menu separator is not appended to the main form's menu list. We only append actual menu items to that list.

Okay, next up we create the "exit" menu item. This code is identical to that for the "dummy1" entry (except, of course, for various names), so it is presented here without further explanation:

```
// Code to create a wMenuItem object:

mov(main,ebx);
wMenuItem_t.create_wMenuItem("menu_exit", ebx,w.MF_STRING, "Exit", &onQuit);
mov( esi, (type mainAppWindow_t [ebx]).menu_exit);
mov(ebx,esi);
this.insertWidget( this.menu_exit);
this.appendMenuItem( this.menu_exit);
```

Now that we've got the three items that go in the file submenu, it's time to constructor that submenu object. Submenu items are `wMenu_t` objects. Other than calling the `wMenu_t.create_wMenu` constructor, this code is quite similar to creating a `wMenuItem_t` object:

```
// Code to create a wSubMenu object:

mov(main,ebx);
wMenu_t.create_wMenu("file_1","file",ebx);
mov( esi, (type mainAppWindow_t [ebx]).file_1);
mov(ebx,esi);
this.insertWidget( this.file_1);
this.appendMenuItem( this.file_1);
```

Okay, at this point we've created four separate menu objects. Now it's time to actually tell Windows how we want these menu items connected together. This is accomplished by calling the Win32 API w.AppendMenu function. The w.AppendMenu procedure has the following arguments:

```
AppendMenu: procedure
(
   hMenu        :dword;    // Handle of menu to attach this item to
   uFlags       :dword;    // Menu flags (w.MF_STRING, w.MF_POPUP, etc.)
   uIDNewItem   :dword;    // Windows ID of new menu item
   lpNewItem    :string    // String to display for menu item.
);
```

The `uIDNewItem` is an interesting situation. This is the one place in HOWL where we have to use the Windows' `objectID` value generated for each object. The following is the code that attaches the `dummy1` menu item to the `file_1` submenu:

```
// Submenu item: dummy1

mov( this.file_1, eax );
mov( (type wMenuItem_t [eax]).handle, eax );
mov( this.dummy1,ecx );
mov( eax, (type wMenuItem_t [ecx]).wBase_private.parentHandle );
mov( this.file_1, eax );
w.AppendMenu
(
    (type wMenu_t [eax]).handle,
    (type wMenuItem_t [ecx]).wMenuItem_private.itemType,
    (type wMenuItem_t [ecx]).wBase_private.objectID,
    (type wMenuItem_t [ecx]).wMenuItem_private.itemString
);
```

Okay, here's the code that appends the menu separator object to the `file_1` menu:

```
// Submenu item: _wMenuSeparatorWidget_

mov( this.file_1, eax );
mov( (type wMenuItem_t [eax]).handle, eax );
mov( this._wMenuSeparatorWidget_,ecx );
mov( eax, (type wMenuItem_t [ecx]).wBase_private.parentHandle );
mov( this.file_1, eax );
w.AppendMenu
(
    (type wMenu_t [eax]).handle,
    (type wMenuItem_t [ecx]).wMenuItem_private.itemType,
    (type wMenuItem_t [ecx]).wBase_private.objectID,
    (type wMenuItem_t [ecx]).wMenuItem_private.itemString
);
```

Next we have the code that appends the exit menu entry to the end of the `file_1` submenu:

```
// Submenu item: menu_exit

mov( this.file_1, eax );
mov( (type wMenuItem_t [eax]).handle, eax );
mov( this.menu_exit,ecx );
mov( eax, (type wMenuItem_t [ecx]).wBase_private.parentHandle );
mov( this.file_1, eax );
w.AppendMenu
(
    (type wMenu_t [eax]).handle,
    (type wMenuItem_t [ecx]).wMenuItem_private.itemType,
    (type wMenuItem_t [ecx]).wBase_private.objectID,
    (type wMenuItem_t [ecx]).wMenuItem_private.itemString
);
```

At this point, we have a complete submenu item (file_1). Now we have to create the main menu item so we can attach our submenu to that main menu. Syntactically, a main menu is just a special case of a submenu. The only difference is that we don't insert the main menu into the form's menu list (because the form's menu list specifies the main menu's items):

```
    // Code to create a wMainMenu object:

    mov( ebx, esi );
    wMenu_t.create_wMenu("mainAppWindow_menu", "main",ebx);
    mov( esi, (type mainAppWindow_t [ebx]).mainAppWindow_menu);
    mov(ebx,esi);
    main.insertWidget( this.mainAppWindow_menu);
```

After creating the main menu, all that is left to do is to attach the `file_1` submenu to the main menu. The following code does this. Note the use of the `w.MF_POPUP` type when attaching a submenu to the main menu:

```
    // Main menu item: file_1

    mov( this.mainAppWindow_menu, eax );
    mov( (type wMenuItem_t [eax]).handle, eax );
    mov( this.file_1,ecx );
    mov( eax, (type wMenuItem_t [ecx]).wBase_private.parentHandle );
    mov( this.mainAppWindow_menu, eax );
    w.AppendMenu
   (
       (type wMenu_t [eax]).handle,
       w.MF_POPUP,
       (type wMenu_t [ecx]).handle,
       (type wMenuItem_t [ecx]).wMenuItem_private.itemString
   );

    mov( this.mainAppWindow_menu, ecx );
    w.SetMenu( this.handle, (type wMenu_t [ecx]).handle );
    w.DrawMenuBar( this.handle );


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

end create_mainAppWindow;
```

The remainder of the code in the *014x_menu1.hla* sample program is identical to that in the *014_menu1.hla* source file. Please see the example in the previous section for that code.