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

#4: Radio Buttons

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

Prerequisites:

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

Source Code:

The source code for the examples appearing in this tutorial are available as part of the HLA Examples download. You'll find the sample code in the Win32/HOWL subdirectories in the unpacked examples download. This particular tutorial uses the files *009_radioset1.hla, 010_radioset2.hla,* and *011_radioset3.hla*. Though this particular document does not describe *009x_radioset1.hla, 010x_radioset2.hla,* and *011x_radioset3.hla,* , you may also find these files of interest when reading through this tutorial.

Radio Buttons:

Checkboxes are a special type of button that have a binary (check/not checked) state associated with them. In a sense, they are quite similar to check box buttons. However, radio buttons generally appear in groups and an application generally allows only a single radio button in a group to be in the on state. Generally, "clicking" on a radio button will toggle the radio button's state on and turn all other radio buttons in the same group to the off state. Here is what a typical set of radio buttons will look like on a form:

The radio button consists of two components on the form: the white circle is the actual radio button (and will contain a dot if in the true state, it will be empty if in the false state).

Because radio buttons are special types of buttons, it should come as no surprise the that HOWL Declarative Language (HDL) syntax for a radio button declaration is nearly identical to that of push buttons (that we looked at in the last chapter). Here is a typical HDL `wRadioButton` declaration:

```
wRadioButton
(
    radioButton1,           // Field name in mainWindow object
    "Radio Button #1",      // Caption for push button
    0,                      // Style
    10,                     // x position
    10,                     // y position
    125,                    // width
    25,                     // height
    onClick1                // "on click" event handler
)
```

Except for the name (`wRadioButton`), this declaration is identical to that of a `wPushButton` object:

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

Like check boxes, there is also a "left text" version of a radio button that puts the caption on the left side of the control's bounding box and the actual radio button on the right side. Here is an example of a `wRadioButtonLT` declaration:

```
wRadioButtonLT
(
    radioButton2,           // Field name in mainWindow object
    "Radio Button #2",      // Caption for push button
    0,                      // Style
    10,                     // x position
    70,                     // y position
    125,                    // width
    25,                     // height
    onClick1                // "on click" event handler
)
```

Unlike check boxes, there are no three-state radio buttons.

Although Windows considers them both buttons, radio buttons are fundamentally different from push buttons. When the user clicks on a push button this is a signal to the application that some action should take place. Radio buttons, on the other hand, typically use clicks to change their state and perform no other action. A typical onClick handler for a HOWL radio button will be the following:

```
proc onClick1:widgetProc;
begin onClick1;

    // Invert the dot in the radio button:

    mov( thisPtr, esi );
    (type wRadioButton_t [esi]).get_check();
    xor( 1, eax );
    and( 1, eax );
    (type wRadioButton_t [esi]).set_check( eax );


end onClick1;
```

The `get_check` method call returns the current state of the radio button (0/false = unset, 1/true = set). The code above will invert the value read by `get_check` (the `xor` instruction does this) and then writes the new boolean value back to the radio button via the `set_check` method call.

Note that the HOWL `wRadioButton` widget does not automatically process button clicks. That is, if you create a `wRadioButton` widget and don't provide an "on-click" handler for that widget, when the user clicks on the radio button it will not automatically switch between states. It is possible to create automatic radio button widgets, where Windows handles the logic of the radio button when the user clicks on it, we'll take a look at automatic radio buttons later in this tutorial. Also note that standard `wRadioButton` objects do not associate in groups. In particular, if you place multiple `wRadioButton` objects on a form, your code is responsible for making sure only one radio button is checked at a time. Later in this tutorial, you'll see how to automate that logic.

Generally, when an application uses a radio button, it calls `get_check` to obtain the state of the button whenever it needs to test that value. This may occur long after the user has clicked on the radio button to change the state (assuming they have clicked on it -- the user could have left the radio button in its initial state).

The first demo program in this tutorial is a small modification of the `007_checkbox3.hla` source file from the previous tutorial -- It simply swaps a pair of radio button objects for the check box objects on the form (plus a few textual changes to say "radio button" rather than "check box"). :

```
// radioSet1-
//
//  This program demonstrates operations on a radio button, including
// simulated button clicks, double clicks, showing and hiding radio buttons,
// enabling and disabling radio buttons, moving radio buttons, and resizing
// radio buttons.

program radioSet1;
#linker( "comdlg32.lib" )
#linker( "comctl32.lib" )

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

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

const
    applicationName := "Radio Buttons #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 onClick1                   :widgetProc; @forward;
proc hideShowRadioButton        :widgetProc; @forward;
proc enableDisableRadioButton   :widgetProc; @forward;
proc moveRadioButton            :widgetProc; @forward;
proc resizeRadioButton          :widgetProc; @forward;
proc onDblClick                 :widgetProc; @forward;
proc onQuit                     :widgetProc; @forward;



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

wForm( mainAppWindow );

    var
        showState   :boolean;
        b1Enabled   :boolean;
```

```
    align(4);

wRadioButton
(
    radioButton1,           // Field name in mainWindow object
    "Radio Button #1 abc",  // Caption for push button
    0,                      // Style
    10,                     // x position
    10,                     // y position
    125,                    // width
    25,                     // height
    onClick1                // "on click" event handler
)


wRadioButtonLT
(
    radioButton2,           // Field name in mainWindow object
    "Radio Button #2 abc",  // Caption for push button
    0,                      // Style
    10,                     // x position
    70,                     // y position
    125,                    // width
    25,                     // height
    onClick1                // "on click" event handler
)


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


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


wPushButton
(
    button4,                // Field name in mainWindow object
    "Move radio button 1",  // Caption for push button
    175,                    // x position
    70,                     // y position
    175,                    // width
```

```
        25,                     // height
        moveRadioButton         // initial "on click" event handler
    )


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


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


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

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

endwForm


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

mainAppWindow_implementation();







// The onDblClick widget proc will handle a double click on button6
// and simulate a single click on radio button1.
```

```
proc onDblClick:widgetProc;
begin onDblClick;

    mov( mainAppWindow.radioButton1, esi );
    (type wRadioButton_t [esi]).click();
    mov( mainAppWindow.radioButton2, esi );
    (type wRadioButton_t [esi]).click();

end onDblClick;



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

proc resizeRadioButton:widgetProc;
begin resizeRadioButton;

    mov( mainAppWindow.radioButton1, esi );
    (type wRadioButton_t [esi]).get_width();
    if( eax = 125 ) then

        stdout.put( "Resizing radio button to width 150" nl );
        (type wRadioButton_t [esi]).set_width( 150 );
        mov( mainAppWindow.radioButton2, esi );
        (type wRadioButton_t [esi]).set_width( 150 );

    else

        stdout.put( "Resizing radio button to width 125" nl );
        (type wRadioButton_t [esi]).set_width( 125 );
        mov( mainAppWindow.radioButton2, esi );
        (type wRadioButton_t [esi]).set_width( 125 );

    endif;

end resizeRadioButton;



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

proc moveRadioButton:widgetProc;
begin moveRadioButton;

    mov( mainAppWindow.radioButton1, esi );
    (type wRadioButton_t [esi]).get_y();
    if( eax = 10 ) then

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

    else

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

    endif;
```

```
end moveRadioButton;



// The enableDisableRadioButton widget proc will hide and show radio button1.

proc enableDisableRadioButton:widgetProc;
begin enableDisableRadioButton;

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

        (type wRadioButton_t [esi]).set_text( "Enable radio button 1" );
        mov( false, mainAppWindow.b1Enabled );
        stdout.put( "Disabling button 1" nl );
        mov( mainAppWindow.radioButton1, esi );
        (type wRadioButton_t [esi]).disable();

    else

        (type wRadioButton_t [esi]).set_text( "Disable radio button 1" );
        mov( true, mainAppWindow.b1Enabled );
        stdout.put( "Enabling button 1" nl );
        mov( mainAppWindow.radioButton1, esi );
        (type wRadioButton_t [esi]).enable();

    endif;

end enableDisableRadioButton;



// The hideShowRadioButton widget proc will hide and show radio button1.

proc hideShowRadioButton:widgetProc;
begin hideShowRadioButton;

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

        (type wRadioButton_t [esi]).set_text( "Hide radio button 1" );
        mov( false, mainAppWindow.showState );
        stdout.put( "Showing button 1" nl );
        mov( mainAppWindow.radioButton1, esi );
        (type wRadioButton_t [esi]).show();

    else

        (type wRadioButton_t [esi]).set_text( "Show radio button 1" );
        mov( true, mainAppWindow.showState );
        stdout.put( "Hiding button 1" nl );
        mov( mainAppWindow.radioButton1, esi );
        (type wRadioButton_t [esi]).hide();

    endif;

end hideShowRadioButton;
```

```
// Here's the onClick handler for the radio button. Invert the state on each click.


proc onClick1:widgetProc;
begin onClick1;

    // Invert the dot in the radio button:

    mov( thisPtr, esi );
    (type wRadioButton_t [esi]).get_check();
    xor( 1, eax );
    and( 1, eax );
    (type wRadioButton_t [esi]).set_check( eax );


end onClick1;



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

proc onQuit:widgetProc;
begin onQuit;

    // Quit the app:

    w.PostQuitMessage( 0 );

end onQuit;





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

method mainAppWindow_t.onCreate;
var
    thisPtr :dword;

begin onCreate;

    mov( esi, thisPtr );

    // Initialize the showState and enableDisableButton data fields:

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

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

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

end onCreate;



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

method mainAppWindow_t.onClose;
begin onClose;

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

    w.PostQuitMessage( 0 );


end onClose;




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

procedure appStart;
begin appStart;

    push( esi );

    // Create the main application window:

    mainAppWindow.create_mainAppWindow
    (
        applicationName,        // Window title
        w.WS_EX_CONTROLPARENT,  // Need this to support TAB control selection
        w.WS_OVERLAPPEDWINDOW,  // Style
        NULL,                  // No parent window
        formX,                  // 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 radioSet1;

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

    w.GetSysColor( w.COLOR_MENU );
    mov( eax, howl.bkgColor_g );
    or( $FF00_0000, eax );
```

```
    mov( eax, howl.transparent_g );
    w.CreateSolidBrush( howl.bkgColor_g );
    mov( eax, howl.bkgBrush_g );

    // Start the HOWL Framework Main Program:

    HowlMainApp();

    // Delete the brush we created earlier:

    w.DeleteObject( howl.bkgBrush_g );

end radioSet1;
```

The HOWL wRadioButton_t and wRadioButtonLT_t Classes

Before moving on to the next example, it's worthwhile to quickly discuss the two classes that HOWL uses to define radio buttons.

The `wRadioButton_t` class is derived from `wCheckable_t` (see the last tutorial for a discussion of the `wCheckable_t` class). Here's the class definition:

```
    wRadioButton_t:
        class inherits( wCheckable_t );

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


        endclass;
```

Unlike the check box classes, you'll notice that the constructor exposes the Windows button `style` argument. Here are typical values you would supply as the `style` argument:

Choose one of these two constant values:

w.BS_RADIOBUTTON	Creates a small circle with text. By default, the text is displayed to the right of the circle. To display the text to the left of the circle, combine this flag with the w.BS_LEFTTEXT style (or with the equivalent w.BS_RIGHTBUTTON style). Use radio buttons for groups of related, but mutually exclusive choices.

w.BS_AUTORADIOBUTTON	Creates a button that is the same as a radio button, except that when the user selects it, Windows automatically sets the button's check state to checked and automatically sets the check state for all other buttons in the same group to unchecked.

Possibly OR'd logically with one or more of the following constants (as appropriate:

w.BS_BOTTOM	Places text at the bottom of the button rectangle.
w.BS_LEFTTEXT	Places text on the left side of the radio button or check box when combined with a radio button or check box style. Same as the w.BS_RIGHTBUTTON style.
w.BS_CENTER	Centers text horizontally in the button rectangle.
w.BS_LEFT	Left-justifies the text in the button rectangle. However, if the button is a check box or radio button that does not have the w.BS_RIGHTBUTTON style, the text is left justified on the right side of the check box or radio button.
w.BS_PUSHLIKE	Makes a button (such as a check box, three-state check box, or radio button) look and act like a push button. The button looks raised when it isn't pushed or checked, and sunken when it is pushed or checked.w.BS_RIGHTRight-justifies text in the button rectangle. However, if the button is a check box or radio button that does not have the w.BS_RIGHTBUTTON style, the text is right justified on the right side of the check box or radio button.
w.BS_RIGHTBUTTON	Positions a radio button's circle or a check box's square on the right side of the button rectangle. Same as the w.BS_LEFTTEXT style.
w.BS_TOP	Places text at the top of the button rectangle.
w.BS_VCENTER	Places text in the middle (vertically) of the button rectangle.
w.WS_GROUP	This constant must be attached to the first button of a radio button group.
w.WS_TABSTOP	This constant should be attached to the first button of a radio button group.

If you supply zero as the `style` argument, or the value does not contain either the w.BS_RADIOBUTTON or w.BS_AUTORADIOBUTTON constants, then HOWL will logically OR whatever value you provide the following constant value for the button style:

```
w.BS_RADIOBUTTON | w.WS_GROUP | w.WS_TABSTOP
```

The `wRadioButtonLT_t` class creates a "left text" radio button. This control has the radio button justified to the right of the bounding rectangle and the text on the left side of the button. Otherwise, `wRadioButtonLT_t` objects are identical to `wRadioButton_t` objects:

```
wRadioButtonLT_t:
    class inherits( wCheckable_t );

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

```
endclass;
```

If you supply zero as the `style` argument, or the value does not contain either the w.BS_RADIOBUTTON or w.BS_AUTORADIOBUTTON constants, then HOWL will logically OR whatever value you provide the following constant value for the button style:

```
w.BS_RADIOBUTTON | w.WS_GROUP | w.WS_TABSTOP
```

In all cases, HOWL will also logically OR the value w.BS_LEFTTEXT with the style constant (to force a left text button).

Another Radio Button Example: RadioButton2

The previous example (*009_radioset1.hla*) has a huge problem: the two radio buttons on the form are not part of a group. That is, you can set or unset the radio buttons independently of one another, which is not a standard user interface behavior for a group of radio buttons. Essentially, those radio buttons behave just like check boxes on the form. In this next example (010_radioset2.hla) we'll take a look at how to tell Windows to treat the radio buttons as a group and have Windows automatically control the behavior of the radio buttons.

The first step in automating a group of radio buttons is to supply the `w.BS_AUTORADIOBUTTON` constant as the style for all the radio buttons in a group. This constant tells Windows to automatically handle the logic of setting and unsetting the radio buttons in a group.

The second thing a program has to do is to tell Windows which radio buttons are part of a group of radio buttons. Under Windows, this is done by assigning the style `w.WS_GROUP` to the first button of a group of radio buttons and not supplying this style to the remaining buttons in the group. To avoid confusion, it's a very good idea to declare/create all your group radio buttons using consecutive HDL declarations (or consecutive calls to the radio button constructors).

If you want to allow the user to move between controls on a form by pressing the TAB key, you should also logically OR in the `w.WS_TABSTOP` constant into the first radio button of a group. You should not attach this constant to all the radio buttons in a group, the standard user interface design suggests using the TAB key to switch between groups of radio buttons (and other controls) and using the up and down arrow keys to move between radio buttons in a single group.

Here are the three HDL declarations for the radio buttons in the *010_radioset2.hla* application:

```
wRadioButton
(
    radioButton1,           // Field name in mainWindow object
    "Radio Button #1 abc",  // Caption for push button
    w.BS_AUTORADIOBUTTON | w.WS_GROUP | w.WS_TABSTOP,
    10,                     // x position
    10,                     // y position
    125,                    // width
    25,                     // height
```

```
        NULL                    // No "on click" event handler
    )


    wRadioButtonLT
    (
        radioButton2,           // Field name in mainWindow object
        "Radio Button #2",      // Caption for push button
        w.BS_AUTORADIOBUTTON,   // Style
        10,                     // x position
        70,                     // y position
        125,                    // width
        25,                     // height
        NULL                    // No "on click" event handler
    )

    wRadioButtonLT
    (
        radioButton3,           // Field name in mainWindow object
        "Radio Button #3",      // Caption for push button
        w.BS_AUTORADIOBUTTON,   // Style
        10,                     // x position
        100,                    // y position
        125,                    // width
        25,                     // height
        NULL                    // No "on click" event handler
    )
```

Notice that none of these button declarations have an on-click event handler associated with them. Windows will automatically handle checking and unchecking the radio buttons. An application program generally doesn't care about when the user checks a radio button. Instead, when an application wants to know the state of the radio button it will call the object's `get_state` method to retrieve the current value of that button.

Here's the complete *010_radioset2.hla* application:

```
// radioSet2-
//
//  This program demonstrates operations on a group of radio buttons, including
// simulated button clicks, double clicks, showing and hiding radio buttons,
// enabling and disabling radio buttons, moving radio buttons, and resizing
// radio buttons.
//
// This program also demonstrate automatically handled radio buttons.

program radioSet2;
#linker( "comdlg32.lib" )
#linker( "comctl32.lib" )

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

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

const
    applicationName := "Radio Buttons #2";
```

```
    formX           := w.CW_USEDEFAULT; // Let Windows position this guy
    formY           := w.CW_USEDEFAULT;
    formW           := 600;
    formH           := 600;



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

proc hideShowRadioButton        :widgetProc; @forward;
proc enableDisableRadioButton   :widgetProc; @forward;
proc moveRadioButton            :widgetProc; @forward;
proc resizeRadioButton          :widgetProc; @forward;
proc onDblClick                 :widgetProc; @forward;
proc onQuit                     :widgetProc; @forward;



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

wForm( mainAppWindow );

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

    wRadioButton
    (
        radioButton1,           // Field name in mainWindow object
        "Radio Button #1 abc",  // Caption for push button
        w.BS_AUTORADIOBUTTON | w.WS_GROUP | w.WS_TABSTOP,
        10,                     // x position
        10,                     // y position
        125,                    // width
        25,                     // height
        NULL                    // No "on click" event handler
    )


    wRadioButtonLT
    (
        radioButton2,           // Field name in mainWindow object
        "Radio Button #2",      // Caption for push button
        w.BS_AUTORADIOBUTTON,   // Style
        10,                     // x position
        70,                     // y position
        125,                    // width
        25,                     // height
        NULL                    // No "on click" event handler
    )

    wRadioButtonLT
    (
        radioButton3,           // Field name in mainWindow object
        "Radio Button #3",      // Caption for push button
        w.BS_AUTORADIOBUTTON,   // Style
        10,                     // x position
```

```
    100,                    // y position
    125,                    // width
    25,                     // height
    NULL                    // No "on click" event handler
)


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


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


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


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


wPushButton
(
    button6,                // Field name in mainWindow object
    "DblClick to Click",    // Caption for push button
```

```
        175,                    // x position
        130,                    // y position
        175,                    // width
        25,                     // height
        NULL                    // no single click handler
    )


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

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

endwForm


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

mainAppWindow_implementation();







// The onDblClick widget proc will handle a double click on button6
// and simulate a single click on radio button1.

proc onDblClick:widgetProc;
begin onDblClick;

    mov( mainAppWindow.radioButton1, esi );
    (type wRadioSetButton_t [esi]).click();

end onDblClick;



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

proc resizeRadioButton:widgetProc;
begin resizeRadioButton;

    mov( mainAppWindow.radioButton1, esi );
    (type wRadioSetButton_t [esi]).get_width();
    if( eax = 125 ) then

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

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

    else

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

    endif;

end resizeRadioButton;



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

proc moveRadioButton:widgetProc;
begin moveRadioButton;

    mov( mainAppWindow.radioButton1, esi );
    (type wRadioSetButton_t [esi]).get_y();
    if( eax = 10 ) then

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

    else

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

    endif;

end moveRadioButton;



// The enableDisableRadioButton widget proc will hide and show radio button1.

proc enableDisableRadioButton:widgetProc;
begin enableDisableRadioButton;

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

        (type wRadioSetButton_t [esi]).set_text( "Enable radio button 1" );
        mov( false, mainAppWindow.b1Enabled );
        stdout.put( "Disabling button 1" nl );
        mov( mainAppWindow.radioButton1, esi );
        (type wRadioSetButton_t [esi]).disable();

    else

        (type wRadioSetButton_t [esi]).set_text( "Disable radio button 1" );
        mov( true, mainAppWindow.b1Enabled );
        stdout.put( "Enabling button 1" nl );
        mov( mainAppWindow.radioButton1, esi );
        (type wRadioSetButton_t [esi]).enable();
```

```
    endif;

end enableDisableRadioButton;



// The hideShowRadioButton widget proc will hide and show radio button1.

proc hideShowRadioButton:widgetProc;
begin hideShowRadioButton;

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

        (type wRadioSetButton_t [esi]).set_text( "Hide radio button 1" );
        mov( false, mainAppWindow.showState );
        stdout.put( "Showing button 1" nl );
        mov( mainAppWindow.radioButton1, esi );
        (type wRadioSetButton_t [esi]).show();

        // Reenable the "double-click" button:

        mov( mainAppWindow.button6, esi );
        (type wPushButton_t [esi]).enable();

    else

        (type wRadioSetButton_t [esi]).set_text( "Show radio button 1" );
        mov( true, mainAppWindow.showState );
        stdout.put( "Hiding button 1" nl );
        mov( mainAppWindow.radioButton1, esi );
        (type wRadioSetButton_t [esi]).hide();

        // Must disable the "double-click" button while
        // radionButton1 is hidden:

        mov( mainAppWindow.button6, esi );
        (type wPushButton_t [esi]).disable();

    endif;

end hideShowRadioButton;







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

proc onQuit:widgetProc;
begin onQuit;

    // Quit the app:

    w.PostQuitMessage( 0 );
```

```
end onQuit;




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

method mainAppWindow_t.onCreate;
var
    thisPtr :dword;

begin onCreate;

    mov( esi, thisPtr );

    // Initialize the showState and enableDisableButton data fields:

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

    // Set up button6's onDblClick handler:

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

end onCreate;



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

method mainAppWindow_t.onClose;
begin onClose;

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

    w.PostQuitMessage( 0 );
```

```
end onClose;




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

procedure appStart;
begin appStart;

    push( esi );

    // Create the main application window:

    mainAppWindow.create_mainAppWindow
    (
        applicationName,        // Window title
        w.WS_EX_CONTROLPARENT,  // Need this to support TAB control selection
        w.WS_OVERLAPPEDWINDOW,  // Style
        NULL,                  // No parent window
        formX,                  // 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 radioSet2;

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

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

    // Start the HOWL Framework Main Program:

    HowlMainApp();

    // Delete the brush we created earlier:

    w.DeleteObject( howl.bkgBrush_g );

end radioSet2;
```

RadioSet Declarations

Generally, a group of radio buttons are contained within a group box. HOWL provides a special declaration that slightly simplifies this common case. A `wRadioSet..endwRadioSet` block of statements will group together a set of radio buttons and surround them with a group box. Within the `wRadioSet..endwRadioSet` statement you can place `wRadioSetButton` and `wRadioSetButtonLT` declarations (and nothing else). Here is a typical `wRadioSet` statement sequence:

```
    wRadioSet
    (
        radioSet1,              // Field name in mainWindow object
        "Radio Button Set",     // Caption for radio set panel
        10,                     // x position
        10,                     // y position
```

```
        200,                    // width
        200,                    // height
        howl.bkgColor_g
    )

        wRadioSetButton
        (
            radioButton1,           // Field name in mainWindow object
            "Radio Button #1 abc",  // Caption for push button
            10,                     // x position
            20,                     // y position
            125,                    // width
            25,                     // height
            NULL                    // No "on click" event handler
        )


        wRadioSetButtonLT
        (
            radioButton2,           // Field name in mainWindow object
            "Radio Button #2",      // Caption for push button
            10,                     // x position
            80,                     // y position
            125,                    // width
            25,                     // height
            NULL                    // No "on click" event handler
        )

        wRadioSetButtonLT
        (
            radioButton3,           // Field name in mainWindow object
            "Radio Button #3",      // Caption for push button
            10,                     // x position
            110,                    // y position
            125,                    // width
            25,                     // height
            NULL                    // No "on click" event handler
        )

    endwRadioSet
```

The `wRadioSet` declaration defines the group box that will surround the radio buttons. This is a rectangular box with a caption overwriting the line on the top line (left justified). Within the `wRadioSet..endwRadioSet` statement, only `wRadioSetButton` and `wRadioSetButtonLT` statements are legal (in particular, note that `wRadioButton` and `wRadioButtonLT` statements are not legal). Notice that you don't supply a style argument when declaring these buttons. HOWL automatically supplies an appropriate `w.BS_AUTORADIOBUTTON` style (plus `w.WS_GROUP` and `w.WS_TABSTOP` styles for the first button in the sequence).

The *011_radioset3.hla* application is a simple extension of the previous applications in this tutorial that uses the `wRadioSet` sequence to create a radio button group. You will notice in this code (i.e., in the declarations above) that there are no on-click event handlers installed. Because these radio buttons are automatically controlled by Windows, there is no need for a on-click event handler.

```
// radioSet3-
```

```
//
//  This program demonstrates operations on a set of radio buttons, including
// simulated button clicks, double clicks, showing and hiding radio buttons,
// enabling and disabling radio buttons, moving radio buttons, and resizing
// radio buttons.
//
// This program also demonstrate automatically handled radio buttons.

program radioSet3;
#linker( "comdlg32.lib" )
#linker( "comctl32.lib" )

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

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

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


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

proc hideShowRadioButton        :widgetProc; @forward;
proc enableDisableRadioButton   :widgetProc; @forward;
proc moveRadioButton            :widgetProc; @forward;
proc resizeRadioButton          :widgetProc; @forward;
proc onDblClick                 :widgetProc; @forward;
proc onQuit                     :widgetProc; @forward;



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

wForm( mainAppWindow );

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

    wRadioSet
    (
        radioSet1,              // Field name in mainWindow object
        "Radio Button Set",     // Caption for radio set panel
        10,                     // x position
        10,                     // y position
        200,                    // width
        200,                    // height
        howl.bkgColor_g
    )

        wRadioSetButton
```

```
    (
        radioButton1,           // Field name in mainWindow object
        "Radio Button #1 abc",  // Caption for push button
        10,                     // x position
        20,                     // y position
        125,                    // width
        25,                     // height
        NULL                    // No "on click" event handler
    )


    wRadioSetButtonLT
    (
        radioButton2,           // Field name in mainWindow object
        "Radio Button #2",      // Caption for push button
        10,                     // x position
        80,                     // y position
        125,                    // width
        25,                     // height
        NULL                    // No "on click" event handler
    )

    wRadioSetButtonLT
    (
        radioButton3,           // Field name in mainWindow object
        "Radio Button #3",      // Caption for push button
        10,                     // x position
        110,                    // y position
        125,                    // width
        25,                     // height
        NULL                    // No "on click" event handler
    )

endwRadioSet


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


wPushButton
(
    button3,                    // Field name in mainWindow object
    "Disable radio button set", // Caption for push button
    250,                        // x position
    40,                         // y position
    175,                        // width
    25,                         // height
    enableDisableRadioButton    // initial "on click" event handler
)
```

```
    wPushButton
    (
        button4,                    // Field name in mainWindow object
        "Move radio button set",        // Caption for push button
        250,                        // x position
        70,                         // y position
        175,                        // width
        25,                         // height
        moveRadioButton             // initial "on click" event handler
    )


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


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


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

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

endwForm


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

mainAppWindow_implementation();
```

```
// The onDblClick widget proc will handle a double click on button6
// and simulate a single click on radio button1.

proc onDblClick:widgetProc;
begin onDblClick;

    mov( mainAppWindow.radioButton1, esi );
    (type wRadioSetButton_t [esi]).click();

end onDblClick;



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

proc resizeRadioButton:widgetProc;
begin resizeRadioButton;

    mov( mainAppWindow.radioSet1, esi );
    (type wRadioSet_t [esi]).get_width();
    if( eax = 200 ) then

        stdout.put( "Resizing radio button to width 175" nl );
        (type wRadioSet_t [esi]).set_width( 175 );

    else

        stdout.put( "Resizing radio button to width 200" nl );
        (type wRadioSet_t [esi]).set_width( 200 );

    endif;

end resizeRadioButton;



// The moveRadioButton widget proc will move radioset
// between y positions 10 and 40.

proc moveRadioButton:widgetProc;
begin moveRadioButton;

    mov( mainAppWindow.radioSet1, esi );
    (type wRadioSet_t [esi]).get_y();
    if( eax = 10 ) then

        stdout.put( "Moving radio set to y-position 40" nl );
        (type wRadioSet_t [esi]).set_y( 40 );

    else

        stdout.put( "Moving radio set to y-position 10" nl );
        (type wRadioSet_t [esi]).set_y( 10 );
```

```
    endif;

end moveRadioButton;



// The enableDisableRadioButton widget proc will hide and show radio set.

proc enableDisableRadioButton:widgetProc;
begin enableDisableRadioButton;

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

        (type wRadioSetButton_t [esi]).set_text( "Enable radio set" );
        mov( false, mainAppWindow.b1Enabled );
        stdout.put( "Disabling button 1" nl );

        mov( mainAppWindow.radioSet1, esi );
        (type wRadioSet_t [esi]).disable();

    else

        (type wRadioSetButton_t [esi]).set_text( "Disable radio set" );
        mov( true, mainAppWindow.b1Enabled );
        stdout.put( "Enabling button 1" nl );

        mov( mainAppWindow.radioSet1, esi );
        (type wRadioSet_t [esi]).enable();

    endif;

end enableDisableRadioButton;



// The hideShowRadioButton widget proc will hide and show radio button1.

proc hideShowRadioButton:widgetProc;
begin hideShowRadioButton;

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

        (type wRadioSetButton_t [esi]).set_text( "Hide radio set" );
        mov( false, mainAppWindow.showState );
        stdout.put( "Showing button 1" nl );

        mov( mainAppWindow.radioSet1, esi );
        (type wRadioSet_t [esi]).show();

        // Reenable the "double-click" button:

        mov( mainAppWindow.button6, esi );
        (type wPushButton_t [esi]).enable();

    else
```

```
        (type wRadioSetButton_t [esi]).set_text( "Show radio set" );
        mov( true, mainAppWindow.showState );
        stdout.put( "Hiding button 1" nl );

        mov( mainAppWindow.radioSet1, esi );
        (type wRadioSet_t [esi]).hide();

        // Must disable the "double-click" button while
        // radionButton1 is hidden:

        mov( mainAppWindow.button6, esi );
        (type wPushButton_t [esi]).disable();

    endif;

end hideShowRadioButton;







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

proc onQuit:widgetProc;
begin onQuit;

    // Quit the app:

    w.PostQuitMessage( 0 );

end onQuit;




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

method mainAppWindow_t.onCreate;
var
    thisPtr :dword;

begin onCreate;

    mov( esi, thisPtr );

    // Initialize the showState and enableDisableButton data fields:

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

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

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

end onCreate;



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

method mainAppWindow_t.onClose;
begin onClose;

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

    w.PostQuitMessage( 0 );


end onClose;




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

procedure appStart;
begin appStart;

    push( esi );

    // Create the main application window:

    mainAppWindow.create_mainAppWindow
    (
        applicationName,        // Window title
        w.WS_EX_CONTROLPARENT,  // Need this to support TAB control selection
        w.WS_OVERLAPPEDWINDOW,  // Style
        NULL,                  // No parent window
```

```
        formX,                  // 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 radioSet3;

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

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

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

    // Start the HOWL Framework Main Program:

    HowlMainApp();

    // Delete the brush we created earlier:

    w.DeleteObject( howl.bkgBrush_g );

end radioSet3;
```