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

#5: Labels and Text

In this fifth tutorial of this series, we'll take a look at implementing label (text) 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 *012_label1.hla* and *013_label2.hla*. Though this particular document does not describe *012x_label1.hla* and *013x_label2.hla,* you may also find these files of interest when reading through this tutorial.

Labels:

Labels are static text appearing on a form. Indeed, in the first tutorial we could have used a `wLabel` declaration to put the text "hello world" in the middle of the form (though placing "Hello World" in the title bar was sufficient for that tutorial). In the HOWL Declarative Language (HDL), you define a wLabel object using the following syntax:

```
wLabel
(
    labelID,
    "label String",
    x,
    y,
    width,
    height,
    style,
    textColor,
```

```
    backgroundColor
)
```

The `labelID` argument is the HLA identifier that HOWL will use to identify this label object on the current form.

The "label string" argument is the actual text that HOWL will display for the label on the form.

The `x, y, width,` and `height` arguments specify the bounding box that will hold the text. If the text string is too large to fit within the bounding box, Windows will truncate those portions of the string that would go outside the box.

The style argument specifies the attributes Windows will apply the the text string. This argument should be a combination of zero or more of the following constants:

w.DT_BOTTOM	Bottom-justifies text. This value must be combined with w.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_NOCLIP	Draws without clipping. Drawing text is somewhat faster when w.DT_NOCLIP is used.
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 w.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_TABSTOP	Sets tab stops. Bits 15-8 (high-order byte of the low-order word) of the uFormat parameter specify the number of characters for each tab. The default number of characters per tab is eight.
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.

Note that the `w.DT_NOCLIP`, and `w.DT_NOPREFIX` values cannot be used with the `w.DT_TABSTOP` value.

The `textColor` argument is an RGB value that specifies the color of the text. Often this is `RGB( 0, 0, 0 )`, which equals 0 (black).

The `backgroundColor` argument specifies the color of the bounding rectangle behind the text. Generally this is set to the back ground color of the form (or whatever control the text is placed over), e.g., `howl.bkgColor_g` in all these tutorial examples.

The HOWL wLabel_t and wFont_t Classes

Before rushing in an looking at some example code, let's pause for a moment and consider the two text-related classes in HOWL. First, let's start with the `wLabel_t` class:

```
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;
```

This class has four (private) data fields. The `caption` field is the actual string that HOWL will draw on the form for a `wLabel_t` object. The `font` field is a pointer to a `wFont_t` object (we'll look at `wFont_t` in a moment) that describes the font to use when drawing the string, or it contains the value NULL if you want HOWL to use the standard system font. The `alignment` field contains the text style (see the w.DT_* constants described earlier). The `foreColor` field contains the foreground color for the text (the background color is inherited from the `wVisual_t` class).

The constructor for `wLabel_t, create_wLabel,` is a standard HOWL-style constructor. The arguments are the following:

wlName: This is the HLA identifier name that the constructor will store in the `_name` data field (inherited from `wBase_t`).
caption: this is the (initial) string for the `wLabel_t` object that Windows will display on the form.
parent: this is the handle of the parent window object on which the label is placed. Generally, this is the handle of the main application's form, though it could be some other widget's handle if the label is placed in a `wContainer_t` object.
x, y, width, and **height**: these arguments specify the bounding box that contains the label.
alignment: this is a combination of one or more of the `w.DT_*` window style constants presented in the previous section.
forecolor: this is the RGB foreground color used to draw the label's text on the form.
bkgColor: this is the RGB background color that HOWL uses to fill the bounding box's rectangle prior to drawing the text.

The `destroy` method is the destructor for `wLabel_t` objects. Generally, destroying a form on which the `wLabel_t` object is placed will automatically call the destructor for a `wLabel_t` object. Applications will only call this method if they dynamically create a `wLabel_t` object and don't ever insert that object into some `wContainer_t` object (such as a form).

The `processMessage` method is private; applications must never directly call this method.

The `getFont` method is the accessor for the private `font` data field. This method returns the value of the `font` field (that is, a pointer to the `wFont_t` object associated with this label) in the EAX register. Note that the return value may be NULL, meaning that the label is using the standard system font.

The `get_caption` method returns the value of the private `caption` data field. This is a pointer to the actual string data that HOWL will draw on the form. Applications must not directly change the value of this string.

The `a_get_caption` method returns a copy of the string pointed at by the `caption` data field. Applications should call this method when they need to manipulate the string data associated with the `wLabel_t` object. It is the caller's responsibility to free this string's data by calling `str.free` when it is done using the string.

The `get_foreColor` method returns the RGB text color for the string. Note: to get the background color, call the `wVisual_t get_bkgColor` method.

The `set_font` method is the mutator for the `font` data field. You should either pass NULL or the address of a `wFont_t` object as this method's argument.

The `set_caption` mutator method changes the label's caption field to the string you supply as the parameter. Note that HOWL will make a copy of this string; so if you change your original string after calling `set_caption` this will not affect the text that HOWL displays for the `wLabel_t` object.

The `set_foreColor` method lets you change the text's foreground color. Note that you can call the `wVisual_t set_bkgColor` method to set the text's background color.

The `wFont_t` class defines a font that you can use for `wLabel_t` (and other text) objects. Here is the definition of this class:

```
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;
```

The `wFont_t` class defines a font for use by the `wLabel_t` object (and other text objects in the system). This class has several private data fields, including the following:

family This data field specifies the font family and is one of the following constants:

w.FF_DECORATIVE	Specifies a novelty font. An example is Old English.
w.FF_DONTCARE	Specifies a generic family name. This name is used when information about a font does not exist or does not matter.
w.FF_MODERN	Specifies a monospace font with or without serifs. Monospace fonts are usually modern; examples include Pica, Elite, and Courier New®.
w.FF_ROMAN	Specifies a proportional font with serifs. An example is Times New Roman.
w.FF_SCRIPT	Specifies a font that is designed to look like handwriting; examples include Script and Cursive.
w.FF_SWISS	Specifies a proportional font without serifs. An example is Arial.

bold	This field specifies whether the font is a boldfaced font.
italic	This field specifies whether the font is an italic font.
underline	This field specifies whether the font is an underlined font.
strikeout	This field specifies whether the font is a strikeout font.
monospaced	This field specifies whether the font is a monospaced (non-proportional) font.
faceName	This string specifies the Windows font name, such as “Times New Roman”. Note that you cannot expect all fonts to be installed on every system. Certain fonts (like “Times New Roman” and “Ariel”) are available on most systems, but you should enumerate the fonts on the system and let the user pick a font rather than blindly assuming a font is available.
size	This is the font size, in points.

Note that the `wFont_t` class provides accessor methods for all of the private data fields but no mutator functions. The only way to (legally) set these data fields is via a constructor call. Once you create a font you cannot change its properties (other than by creating a new font with the properties you want).

The constructor for wFont_t, create_wFont, contains one argument for each of the private data fields plus the `wfName` and `parentHandle` arguments. The `wfName` field is a string holding the HLA identifier you're going to use to hold the `wFont_t` object (pointer); the constructor initializes the `_name` field with this string's value. The `parentHandle` field is the handle of the main form for your application. HOWL uses this handle to compute the font's size in points based on the resolution of the display device. Just supply the handle value for the main form as this argument.

The `destroy` method is the destructor for a `wForm_t` object. Generally, destroying a form on which the `wFont_t` object is attached will automatically call the destructor for a `wForm_t` object. Applications will only call this method if they dynamically create a `wForm_t` object and don't ever insert that object into some `wContainer_t` object (such as a form).

You will notice that the `wLabel_t` constructor does not let you specify a form object as one of its arguments. By default, the `wLabel_t` constructor uses the system font (that is, it initializes its `font` data field with NULL). If you want to draw a label object with a font other than the system font, you'll need to create a new font and then use the `wLabel_t set_font` mutator to change the label's font on the display. An example of this will appear in this tutorial.

The Label1 Application

With a discussion of the `wLabel_t` and `wFont_t` classes out of the way, we can now get down to the business of looking at our first sample program that uses the `wLabel_t` class: *012_label.hla*. This sample program demonstrates placing a label on a form and doing the usual (for these tutorials) manipulations on that label: moving it, resizing the bounding box, hiding/showing the label, plus changing its color and changing the text associated with the label. Here are the HDL statements that define this form:

```
wForm( mainAppWindow );

    var
        showState   :boolean;
        align(4);

    wLabel
    (
        label1,
        originalString,
        10,
        10,
        200,
        200,
        w.DT_LEFT | w.DT_WORDBREAK,
        RGB( 0, 0, 0 ),
        howl.bkgColor_g
    )

    wPushButton
    (
        button2,                    // Field name in mainWindow object
        "Hide text",                // Caption for push button
        250,                        // x position
        10,                         // y position
        175,                        // width
```

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


wPushButton
(
    button3,                    // Field name in mainWindow object
    "Move text",                // Caption for push button
    250,                        // x position
    40,                         // y position
    175,                        // width
    25,                         // height
    moveText                    // initial "on click" event handler
)


wPushButton
(
    button4,                    // Field name in mainWindow object
    "Resize text",              // Caption for push button
    250,                        // x position
    70,                         // y position
    175,                        // width
    25,                         // height
    resizeText                  // initial "on click" event handler
)


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


wPushButton
(
    button6,                    // Field name in mainWindow object
    "Change text",              // Caption for push button
    250,                        // x position
    130,                        // y position
    175,                        // width
    25,                         // height
    changeText                  // initial "on click" event handler
)




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

wPushButton
(
```

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

endwForm
```

Note that the `wLabel` statement specifies the style `(w.DT_LEFT | w.DT_WORDBREAK)` and the bounding box is rather tall (multiple lines) at 200 pixels. We're going to feed this label object some long strings and Windows will automatically break up these long strings into multi-line text displays, breaking the lines at word boundaries (rather than splitting words in half). The text is also going to be left-justified in the bounding box.

The initial string for the label object turns out to be "This is a somewhat long string that will wrap around in the box". As just noted, Windows will split up the string into multiple lines that will fit into the 200x200 pixel bounding box. Most of the buttons on this form do usual things (for this tutorial series) and it isn't worth commenting on them. The `button6` object's on-click event handler, however, is new and is worth looking at here -- it changes the text associated with the label. There is the `changeText` widgetProc:

```
proc changeText:widgetProc;
var
    aText   :string;
    theText :string;

readonly
    newText :string :=
        "This is the new text to be assigned to the label1 object. "
        "This one is much longer than the original string "
        "and it will nearly fill up the entire bounding box "
        "(large form) when this text is selected into the "
        "wLabel_t object on the form. It is too long to fit "
        "entirely into the smaller bounding box for the text object.";

begin changeText;

    mov( mainAppWindow.label1, esi );
    (type wLabel_t [esi]).a_get_caption();
    mov( eax, aText );
    (type wLabel_t [esi]).get_caption();
    mov( eax, theText );

    // Sanity check on the two functions:

    assert( str.eq( theText, aText ));

    if( str.eq( theText, originalString ) ) then

        (type wLabel_t [esi]).set_caption( newText );

    else
```

```
        (type wLabel_t [esi]).set_caption( originalString );

    endif;

    // Must free the storage allocated by a_get_caption:

    str.free( aText );

end changeText;
```

This function demonstrates calling both the `get_text` and `a_get_text` methods (though doing so is a bit overkill for this widgetProc). This widgetProc checks the label's current string to see if it matches the original string; if it does, it changes the caption to the value of the `newText` string. If the current caption value is not the original string, then this code sets it back to the original string value.

Another operation worth looking at is the on-click event handler for `button5`: `colorText`. Here is the code for that widgetProc:

```
proc colorText:widgetProc;
begin colorText;

    mov( mainAppWindow.label1, esi );

    (type wLabel_t [esi]).get_foreColor();
    if( eax = RGB( 0, 0, 0 ) ) then

        stdout.put( "Changing label color to yellow/red" nl );
        (type wLabel_t [esi]).set_foreColor( RGB( 255, 255, 0 ) );
        (type wLabel_t [esi]).set_bkgColor( RGB( 255, 0, 0 ) );

    else

        stdout.put( "Changing label color to black/gray" nl );
        (type wLabel_t [esi]).set_foreColor( RGB( 0, 0, 0 ) );
        (type wLabel_t [esi]).set_bkgColor( howl.bkgColor_g );

    endif;

end colorText;
```

This function gets the foreground color for the label object. If it's black (`RGB( 0, 0, 0)`) then this function sets the foreground/background colors to yellow and red. If it's not black (meaning it's probably yellow and red), then this method sets the colors to black/`howl.bkgColor_g`.

Here is the complete *012_label1.hla* source file:

```
// label1-
//
//  This program demonstrates the use of labels on a form.

program label1;
#linker( "comdlg32.lib" )
#linker( "comctl32.lib" )

?@NoDisplay     := true;
```

```
?@NoStackAlign  := true;

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

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

static
    align( 4 );

    originalString  :string :=
        "This is a somewhat long string that will wrap around in the box";


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

proc hideShowText       :widgetProc; @forward;
proc moveText           :widgetProc; @forward;
proc resizeText         :widgetProc; @forward;
proc colorText          :widgetProc; @forward;
proc changeText         :widgetProc; @forward;
proc onQuit             :widgetProc; @forward;



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

wForm( mainAppWindow );

    var
        showState   :boolean;
        align(4);


    wLabel
    (
        label1,
        originalString,
        10,
        10,
        200,
        200,
        w.DT_LEFT | w.DT_WORDBREAK,
        RGB( 0, 0, 0 ),
        howl.bkgColor_g
    )

    wPushButton
    (
        button2,                    // Field name in mainWindow object
        "Hide text",                // Caption for push button
        250,                        // x position
        10,                         // y position
```

```
    175,                        // width
    25,                         // height
    hideShowText                // initial "on click" event handler
)


wPushButton
(
    button3,                    // Field name in mainWindow object
    "Move text",                // Caption for push button
    250,                        // x position
    40,                         // y position
    175,                        // width
    25,                         // height
    moveText                    // initial "on click" event handler
)


wPushButton
(
    button4,                    // Field name in mainWindow object
    "Resize text",              // Caption for push button
    250,                        // x position
    70,                         // y position
    175,                        // width
    25,                         // height
    resizeText                  // initial "on click" event handler
)


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


wPushButton
(
    button6,                    // Field name in mainWindow object
    "Change text",              // Caption for push button
    250,                        // x position
    130,                        // y position
    175,                        // width
    25,                         // height
    changeText                  // initial "on click" event handler
)




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

wPushButton
```

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

endwForm


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

mainAppWindow_implementation();







// The changeText widget proc will change the label's text.

proc changeText:widgetProc;
var
    aText   :string;
    theText :string;

readonly
    newText :string :=
        "This is the new text to be assigned to the label1 object. "
        "This one is much longer than the original string "
        "and it will nearly fill up the entire bounding box "
        "(large form) when this text is selected into the "
        "wLabel_t object on the form. It is too long to fit "
        "entirely into the smaller bounding box for the text object.";

begin changeText;

    mov( mainAppWindow.label1, esi );
    (type wLabel_t [esi]).a_get_caption();
    mov( eax, aText );
    (type wLabel_t [esi]).get_caption();
    mov( eax, theText );

    // Sanity check on the two functions:

    assert( str.eq( theText, aText ));

    if( str.eq( theText, originalString ) ) then

        (type wLabel_t [esi]).set_caption( newText );

    else

        (type wLabel_t [esi]).set_caption( originalString );
```

```
    endif;

    // Must free the storage allocated by a_get_caption:

    str.free( aText );



end changeText;



// The colorText widget proc will change the foreground and background color.

proc colorText:widgetProc;
begin colorText;

    mov( mainAppWindow.label1, esi );

    (type wLabel_t [esi]).get_bkgColor();   // Call just to verify it works.

    (type wLabel_t [esi]).get_foreColor();
    if( eax = RGB( 0, 0, 0 ) ) then

        stdout.put( "Changing label color to yellow/red" nl );
        (type wLabel_t [esi]).set_foreColor( RGB( 255, 255, 0 ) );
        (type wLabel_t [esi]).set_bkgColor( RGB( 255, 0, 0 ) );

    else

        stdout.put( "Changing label color to black/gray" nl );
        (type wLabel_t [esi]).set_foreColor( RGB( 0, 0, 0 ) );
        (type wLabel_t [esi]).set_bkgColor( howl.bkgColor_g );

    endif;

end colorText;



// The resizeText widget proc will resize label1 between widths 150 and 200.

proc resizeText:widgetProc;
begin resizeText;

    mov( mainAppWindow.label1, esi );
    (type wLabel_t [esi]).get_width();
    if( eax = 200 ) then

        stdout.put( "Resizing label to width/height 150" nl );
        (type wLabel_t [esi]).resize( 150, 150 );

    else

        stdout.put( "Resizing label to width/height 200" nl );
        (type wLabel_t [esi]).resize( 200, 200 );

    endif;
```

```
end resizeText;



// The moveText widget proc will move label
// between y positions 10 and 40.

proc moveText:widgetProc;
begin moveText;

    mov( mainAppWindow.label1, esi );
    (type wLabel_t [esi]).get_y();
    if( eax = 10 ) then

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

    else

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

    endif;

end moveText;



// The hideShowText widget proc will hide and show label1.

proc hideShowText:widgetProc;
begin hideShowText;

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

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

        mov( mainAppWindow.label1, esi );
        (type wLabel_t [esi]).show();

    else

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

        mov( mainAppWindow.label1, esi );
        (type wLabel_t [esi]).hide();

    endif;

end hideShowText;
```

```
// 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;

    // Initialize the showState data field:

    mov( false, this.showState );



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 label1;

    // 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 label1;
```

A wLabel Example using a Font

The previous example used the system font to display the wLabel_t object on the form. This tutorial will demonstrate how to create a font and switch the text between that font and the system font during program execution.

The application code for this new program is very similar to the previous tutorial. The first couple of differences occur in the `wForm..endwForm` sequence:

```
wForm( mainAppWindow );
```

```
    var
        theFont     :wFont_p;
        showState   :boolean;
        align(4);
          .
          .
          .
       << same declarations as found in 012_label1.hla >>
          .
          .
          .
    wPushButton
    (
        button7,                    // Field name in mainWindow object
        "Change font",              // Caption for push button
        250,                        // x position
        160,                        // y position
        175,                        // width
        25,                         // height
        changeFont                  // initial "on click" event handler
    )
          .
          .
          .

endwForm
```

The new `theFont` data field is going to hold a pointer to the `wFont_t` object we're going to create. The new `button7` object will be responsible for changing between the system font and the new font. The `changeFont` widgetProc, that handles this operation, is

```
proc changeFont:widgetProc;
begin changeFont;

    mov( mainAppWindow.label1, esi );
    (type wLabel_t [esi]).get_font();
    if( eax = NULL ) then

        (type wLabel_t [esi]).set_font( mainAppWindow.theFont );

    else

        (type wLabel_t [esi]).set_font( NULL );

    endif;

end changeFont;
```

This code is fairly straight-forward. If the `label1` object's font field is currently NULL, then this code invokes the `set_font` method to change it to the new font we've created (specified by the `theFont` data field). If the font is not NULL (meaning it points at the new font we've created), then this code sets the font field to NULL (the system font).

The last major difference between this code and the previous example is that we have to create the font and initialize the form's `theFont` data field. This application places that initialization code in the `onCreate` method for the form. Here's that code:

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

begin onCreate;

    mov( esi, thisPtr );

    // Initialize the showState data field:

    mov( false, this.showState );

    wFont_t.create_wFont
    (
        "theFont",          // name on form.
        this.handle,        // Parent handle
        "Times New Roman",  // typeface name
        w.FF_ROMAN,         // font family
        14,                 // size
        false,              // bold
        false,              // italic
        false,              // underline
        false,              // strikeout
        false               // monospaced
    );
    mov( esi, eax );
    mov( thisPtr, esi );
    mov( eax, this.theFont );
    this.insertWidget( eax );

end onCreate;
```

Note that the `onCreate` method calls this.insertWidget to attach the `wFont_t` object we create to the main form's widget list. This is done so that when the application quits it will automatically call the destructor for the `wFont_t` object and we don't have to manually supply the call to the `wFont_t` destructor to do this.

Here's the complete code for the *013_label2.hla* application:

```
// label2-
//
//  This program demonstrates the use of labels and fonts on a form.

program label1;
#linker( "comdlg32.lib" )
#linker( "comctl32.lib" )

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

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

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

static
    align( 4 );

    originalString  :string :=
        "This is a somewhat long string that will wrap around in the box";


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

proc hideShowText       :widgetProc; @forward;
proc moveText           :widgetProc; @forward;
proc resizeText         :widgetProc; @forward;
proc colorText          :widgetProc; @forward;
proc changeText         :widgetProc; @forward;
proc changeFont         :widgetProc; @forward;
proc onQuit             :widgetProc; @forward;



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

wForm( mainAppWindow );

    var
        theFont     :wFont_p;
        showState   :boolean;
        align(4);


    wLabel
    (
        label1,
        originalString,
        10,
        10,
        200,
        200,
        w.DT_LEFT | w.DT_WORDBREAK,
        RGB( 0, 0, 0 ),
        howl.bkgColor_g
    )

    wPushButton
    (
        button2,                    // Field name in mainWindow object
        "Hide text",                // Caption for push button
        250,                        // x position
        10,                         // y position
        175,                        // width
        25,                         // height
```

```
    hideShowText                // initial "on click" event handler
)


wPushButton
(
    button3,                    // Field name in mainWindow object
    "Move text",                // Caption for push button
    250,                        // x position
    40,                         // y position
    175,                        // width
    25,                         // height
    moveText                    // initial "on click" event handler
)


wPushButton
(
    button4,                    // Field name in mainWindow object
    "Resize text",              // Caption for push button
    250,                        // x position
    70,                         // y position
    175,                        // width
    25,                         // height
    resizeText                  // initial "on click" event handler
)


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


wPushButton
(
    button6,                    // Field name in mainWindow object
    "Change text",              // Caption for push button
    250,                        // x position
    130,                        // y position
    175,                        // width
    25,                         // height
    changeText                  // initial "on click" event handler
)


wPushButton
(
    button7,                    // Field name in mainWindow object
    "Change font",              // Caption for push button
    250,                        // x position
    160,                        // y position
    175,                        // width
```

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



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

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

endwForm


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

mainAppWindow_implementation();







// The changeFont widget proc will change the label's font.

proc changeFont:widgetProc;
begin changeFont;

    mov( mainAppWindow.label1, esi );
    (type wLabel_t [esi]).get_font();
    if( eax = NULL ) then

        (type wLabel_t [esi]).set_font( mainAppWindow.theFont );

    else

        (type wLabel_t [esi]).set_font( NULL );

    endif;

end changeFont;



// The changeText widget proc will change the label's text.

proc changeText:widgetProc;
var
```

```
    aText   :string;
    theText :string;

readonly
    newText :string :=
        "This is the new text to be assigned to the label1 object. "
        "This one is much longer than the original string "
        "and it will nearly fill up the entire bounding box "
        "(large form) when this text is selected into the "
        "wLabel_t object on the form. It is too long to fit "
        "entirely into the smaller bounding box for the text object.";

begin changeText;

    mov( mainAppWindow.label1, esi );
    (type wLabel_t [esi]).a_get_caption();
    mov( eax, aText );
    (type wLabel_t [esi]).get_caption();
    mov( eax, theText );

    // Sanity check on the two functions:

    assert( str.eq( theText, aText ));

    if( str.eq( theText, originalString ) ) then

        (type wLabel_t [esi]).set_caption( newText );

    else

        (type wLabel_t [esi]).set_caption( originalString );

    endif;

    // Must free the storage allocated by a_get_caption:

    str.free( aText );



end changeText;



// The colorText widget proc will change the foreground and background color.

proc colorText:widgetProc;
begin colorText;

    mov( mainAppWindow.label1, esi );

    (type wLabel_t [esi]).get_bkgColor();   // Call just to verify it works.

    (type wLabel_t [esi]).get_foreColor();
    if( eax = RGB( 0, 0, 0 ) ) then

        stdout.put( "Changing label color to yellow/red" nl );
        (type wLabel_t [esi]).set_foreColor( RGB( 255, 255, 0 ) );
        (type wLabel_t [esi]).set_bkgColor( RGB( 255, 0, 0 ) );
```

```
    else

        stdout.put( "Changing label color to black/gray" nl );
        (type wLabel_t [esi]).set_foreColor( RGB( 0, 0, 0 ) );
        (type wLabel_t [esi]).set_bkgColor( howl.bkgColor_g );

    endif;

end colorText;



// The resizeText widget proc will resize label1 between widths 150 and 200.

proc resizeText:widgetProc;
begin resizeText;

    mov( mainAppWindow.label1, esi );
    (type wLabel_t [esi]).get_width();
    if( eax = 200 ) then

        stdout.put( "Resizing label to width/height 150" nl );
        (type wLabel_t [esi]).resize( 150, 150 );

    else

        stdout.put( "Resizing label to width/height 200" nl );
        (type wLabel_t [esi]).resize( 200, 200 );

    endif;

end resizeText;



// The moveText widget proc will move label
// between y positions 10 and 40.

proc moveText:widgetProc;
begin moveText;

    mov( mainAppWindow.label1, esi );
    (type wLabel_t [esi]).get_y();
    if( eax = 10 ) then

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

    else

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

    endif;

end moveText;
```

```
// The hideShowText widget proc will hide and show label1.

proc hideShowText:widgetProc;
begin hideShowText;

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

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

        mov( mainAppWindow.label1, esi );
        (type wLabel_t [esi]).show();

    else

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

        mov( mainAppWindow.label1, esi );
        (type wLabel_t [esi]).hide();

    endif;

end hideShowText;







// 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 data field:

    mov( false, this.showState );

    wFont_t.create_wFont
    (
        "theFont",          // name on form.
        this.handle,        // Parent handle
        "Times New Roman",  // typeface name
        w.FF_ROMAN,         // font family
        14,                 // size
        false,              // bold
        false,              // italic
        false,              // underline
        false,              // strikeout
        false               // monospaced
    );
    mov( esi, eax );
    mov( thisPtr, esi );
    mov( eax, this.theFont );
    this.insertWidget( eax );

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 label1;

    // 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 label1;
```