IonicWind Software

IWBasic => Create a Custom Control => Topic started by: LarryMc on July 24, 2011, 01:17:42 PM

Title: 1. Introduction
Post by: LarryMc on July 24, 2011, 01:17:42 PM

Introduction

So, here it is, 2011, and I have finished another big project. I need another project to work on. But this time things are a little different. I just spent 18 months on a project that generated little to no activity in the forums other than my postings of where I was at and how many lines of code I had generated. I knew at the time that it was a rather poor attempt to let members (and visitors) know that the light was still on; someone was indeed around who was programming actively with the user base in mind. But, since I absolutely refuse to pre-sale an unfinished product, there was nothing for members to "play" with to wet their appetites.

This time, I wanted a project that:
-Benefits the forum members
-Provides insight to how windows functions in an easy to understand manner.
-Allows for frequent updates of potentially useful information
-Allows for timely feedback from interested members
-Allows members to ask pertinent questions as the project progresses thus impacting its content
-Ultimately results in members creating their own custom controls and sharing

This time, the code and its generation is not the driving force of the project. Instead, providing an understanding of how things work is the primary concern; followed closely by explaining the process used. The actual code is used to merely demonstrate how different portions are actually implemented.

I can offer no estimate of how often I will post updates. Therefore I can't say when it will be finished. My intent is to work on it on a daily basis. But this is akin to writing a help manual and most members know how I feel about that.

LarryMc
"All I like is finishing..."

IonicWind Software

IWBasic => Create a Custom Control => Topic started by: LarryMc on July 24, 2011, 02:43:37 PM

Title: 2. Scope
Post by: LarryMc on July 24, 2011, 02:43:37 PM

Scope

If the reader is expecting a doctorate's thesis on all aspects of creating custom controls for the Windows OS then they are looking in the wrong place.

This is an explanation of custom controls written by an amateur for amateurs. The plan is to explain, in as simple terms as possible, my concept of how windows and controls work. I say "my concept" because I admit that I sometimes use the wrong word to describe a function or feature.

I do not plan on typing every word only after referring to a Microsoft glossary/dictionary. I hope to use words that mean basically the same to everyone in order to give the reader a "feel" for what is going on as opposed to a precise technical understanding. Some things I ask the reader to accept on faith; that's what we do with windows all the time anyway.

With the above in mind, the reader should also remember that I am not a prolific writer. I'm sure my writing will remind the reader of that on a regular basis. The goal will always be to give the reader the necessary words to gain a comfortable way to understand what is going on and why some of the steps discussed are required.

My intent is also to describe how IWBasic handles windows and controls first and then tie that to how the Windows OS handles windows.

That discussion alone should help the more amateur readers understand why they are required to do some of the things they have to do when writing a windows/dialog based program.

All of the discussions to this point will be accomplished without any actual custom control code. However, IWBasic code will be used along with some of the Window's API commands.

If the reader decides to create a custom control they will probably want to share their creation. This will require the creation of a library. Therefore I intend to have a section that discusses the different types of libraries

There will be a section on "registering classes" and what that really means in the grand scheme of things.

Using all that is learned from the above, the required components of custom controls will be discussed. This will be closely followed by a "specification" for what we want our custom control to do. As a pattern (for learning purposes) we will use my gage control.

That will be followed by a discussion of how to set up a development environment to work on our project and test it as we go. This will show how an application

communicates with the custom control.

At each step of the development process the reader will be provided with code snippets (with explanations) to demonstrate what is being done and why. At appropriate junctures the reader will be provided with compilable code and screen shots of the correct results in order for the user to experiment with and study further.

When the gage control development is completed there will be an explanation of how to modify the contents of the development environment to a distributable form with examples.

As all the above is discussed, notice will be given when there are multiple techniques that could have been used and why the one used was picked.

At the end, the reader will have all the source code to completely rebuild the gage control library.

Note: For the original gage library (the one currently available on the forum) I used the standard windows graphics functions. For this exercise, all the graphics functions have been converted over to GDI+. So, although this tutorial is not devoted to GDI+ it does show enough about how it is implemented and used with some of its simplest commands. This should be enough for anyone who is interested to use as a basis for further study.

Additional note: I'm developing this tutorial in the same software that I use for help files. My intention is to offer this tutorial in eBook format at its completion for a small charge. Source code, and especially screenshots, will appear more organized in the eBook. Images will simply be attachments here.

IonicWind Software

IWBasic => Create a Custom Control => Topic started by: LarryMc on July 25, 2011, 06:27:33 PM

Title: 3. IWBasic Windows
Post by: LarryMc on July 25, 2011, 06:27:33 PM

In order to really understand where we are going with custom controls it is important to understand why we do some of the things we do with windows. The following discussion uses portions of the IWBasic User's Guide.

Opening a window in IWBasic is extremely easy. We use the following command line:

Code:

```
OPENWINDOW variable, left, top, width, height, flags, parent, title, handler
```

First and foremost, we must understand that this is simply a call to a function (a subroutine that returns a value) that is sent nine parameters (pieces of information) to use to do whatever the subroutine has to do to open our window. Let's look at those parameters.

variable

This identifies this specific window. Any given program may have many windows. That being the case the uniqueness of this variable name is how we tell them apart. This is a special type of variable called a User Defined Type (UDT). In this case it is a little of a misnomer since the User doesn't define this one; it is predefined by IWBasic. The fundamental characteristic of UDTs is that they are single variables made up of multiple pieces of data. This particular UDT is of type WINDOW. This designation is purely an arbitrary name that was assigned to it when the language was created. Outside of IWBasic, the Windows OS has no idea what it is or what it is used for.

The structure of this WINDOW UDT is defined as:

Code:

```
TYPE WINDOW
        DEF hWnd as UINT
        DEF hBitmap as UINT
        DEF hFont as UINT
        DEF hPen as UINT
        DEF hClient as UINT
        DEF m_pos as POINT
        DEF m_iBkMode as INT
        DEF m_iROP2 as INT
        DEF m_nPenStyle as INT
        DEF m_nPenWidth as INT
        DEF m_cBack as UINT
        DEF m_cWindow as UINT
        DEF m_bAutoDraw as INT
        DEF m_bTabEnable as UINT
        DEF m_hCursor as UINT
        DEF m_hBackDC as UINT
        DEF m_hCacheDC as UINT
        DEF m_nCacheCount as INT
        DEF m_bDialog as INT
        DEF m_pDlgTemplate as POINTER
        DEF m_pIB2DScreen as POINTER
        DEF m_pIB2DSurface as POINTER
```

```
        DEF hProcedure as UINT
        DEF m_pBrowser as POINTER
        DEF m_hPrintDC as UINT
ENDTYPE
```

It is beyond the scope of this tutorial to describe the purpose of each element of the UDT structure. However, there are several things that should be noted from looking at the above.

- It requires that a lot of information be generated to create an IWBasic window.
- We, as the users, are not required to provide any of that information other than a name for the UDT variable.
-The OPENWINDOW subroutine must be dealing with all that for us.

Later we will be referring back to this UDT.

left, top, width, height

Pretty straight-forward. All four values are in pixels. Left/Top are relative to the upper left corner of the User's screen.
The point here being that we're defining how big the window is and where it is located.

flags

Normally referred to as Style flags. This is a single UINT whose individual bits each have some specific impact on the appearance or functionality of the window. Usually, to make things easier to read, each bit is given a CONST definition representing one bit and various bits are or'd together with the' |' operator.
An example would be:

Code:

```
  @MINBOX|@MAXBOX|@SIZE
```

parent

The WINDOWS variable that identifies the parent of the window being created, if there is one. If there is no parent then a NULL or 0 is entered.

title

The text that will appear in the caption bar at the top of the window. If the window has no caption bar (style flag option) or no text is desired "" is entered. Titles are not mandatory.

handler

This entry is a pointer to a message handler subroutine created by the User. Each window has to have a handler routine to do what - handle its messages. Message Handlers is a whole separate subject covered in the next section.

IonicWind Software

IWBasic => Create a Custom Control => Topic started by: LarryMc on July 26, 2011, 08:21:23 AM

Title: 4a. Messages and Message Handlers
Post by: LarryMc on July 26, 2011, 08:21:23 AM

Part A

Generally, in a console based program things progress in a predefined path from point A to point B. The simplest and most straight forward programs start, perform a series of tasks, and then stop. Of course the program can be made more elaborate with loops and if-blocks. But execution pretty much moves from instruction to instruction in a steady manner. It is with these types of console programs, back in the DOS OS days, that we were exposed to our first hint of a rudimentary event driven program. Consider the two following examples:

Code:
```
OPENCONSOLE
PRINT "Press any key to quit"
DO: UNTIL INKEY$ <> ""
CLOSECONSOLE
END
```

In the above example the program is in a loop waiting for an event to happen. The event is the pressing of any key on the keyboard. To be more precise, the loop is waiting for any one of approximately 100 possible events to occur. That is because the pressing of a specific key is a different event then the pressing of any other key.

Consider the second example:

Code:
```
OPENCONSOLE
PRINT "Press Q to quit"
DO: UNTIL INKEY$ = "Q"
CLOSECONSOLE
END
```

In this example we are in a loop until a single, specific event occurs; the pressing of the 'Q' key. Let's expand this example.

Code:
```
STRING A
OPENCONSOLE
PRINT "Press Q to quit"
DO
   A = INKEY$
   SELECT A
      CASE "A"
         MySubA()
      CASE "B"
         MySubB()
      CASE "C"
         MySubC()
   ENDSELECT
UNTIL A = "Q"
CLOSECONSOLE
END
```

Again, we are still in a loop waiting for an event to occur. Think of INKEY$ as

assigning an event identifier (message) to the event and placing that in the variable A. The SELECT/ENDSELECT block is then used to respond to certain events by executing code specific to that event. In this case, by calling a subroutine.

Two important things are happening in this example that must be understood. The service provided by the INKEY$ function (identifying the events for us) is done without any regard to whether or not we will respond to that event. Also, we can pick which events we want to respond to.

The above is a very, very crude example of a message handler. But it does demonstrate some important concepts. The main problem with the loop described above is that when that loop is entered there is nothing else anywhere in the application that can be executed.

Before proceeding, we need to clarify some terminology. Message handler, event handler, and handler all refer to the same thing in the context of this tutorial. Message, event id, notification, notification message all refer to the same thing; the identity of some event that has occurred.

IonicWind Software

IWBasic => Create a Custom Control => Topic started by: LarryMc on July 28, 2011, 08:08:31 AM

Title: 4b. Messages and Message Handlers
Post by: LarryMc on July 28, 2011, 08:08:31 AM

Part B

Now we turn our attention back to windows based programs. Any program that uses windows automatically, by its very nature, becomes an event driven program. When we create our windows application it has to coexist with other windows based applications. Your PC desktop is a windows application, as is windows explorer and any other application you might have running.

Consider, for example, that you have your prize application open along with windows explorer. You only created your application; not the other. Assume you have been using your application (it is on top of the other) and you want to locate a file in windows explorer. You click on the explorer window. The operating system responds to that click event by sending a message to the explorer window which ultimately causes it to be drawn on top. A message is also sent to your application telling it that it is no longer on top and don't redraw its window where it is covered. The operating system was able to accomplish that by recognizing what window contains the pixel on the screen that the cursor was on when the mouse button was clicked. It then sends the various event messages to all the required message handlers.

The operating system system contains 1,000s of messages. They indicate a wide range of events. Examples include events due to user interaction, network traffic, system processing, and timer activity, Messages can even indicate that an event is starting to occur, occurring, starting to finish occurring, and has finished occurring. And remember, from earlier, these messages are nothing more than a numeric value representing a specific event.

From the IWBasic Windows section we learned that we had to specify a handler routine for our window when we created it. Take the following example:
ex_4.iwb
Code:

```
DEF w1 as WINDOW
OPENWINDOW w1,0,0,350,350,@MINBOX|@MAXBOX|@SIZE,NULL,"Simple Window",&main
WAITUNTIL w1 = 0
END

SUB main
        IF @MESSAGE = @IDCLOSEWINDOW
                CLOSEWINDOW w1
        ENDIF
        RETURN 0
ENDSUB
```

The above is the simplest windows program you can write in IWBasic. If you compile it to a Windows target and then execute it, a window with a white background will appear in the upper left corner of your screen. It can be minimized, maximized, resized, repositioned and closed.

Let's examine each line of the program.

Code:

```
DEF w1 as WINDOW
```

We define a variable, w1, as being UDT type WINDOW as previously described.

———————————————

Code:

```
OPENWINDOW w1,0,0,350,350,@MINBOX|@MAXBOX|@SIZE,NULL,"Simple Window",&main
```

We create our window and identify our handler subroutine as 'main'.

———————————————

Code:

```
WAITUNTIL w1 = 0
```

When a window is created in IWBasic the first element of the WINDOWS UDT (hWnd) is assigned the handle number that the windows OS assigned to that specific window when it was created. This handle is used by the OS to identify what message goes to what window. More about this later. For now suffice it to say that w1 will be some non-zero value when the window is created. The WAITUNTIL command is a loop that waits for the window to close as signified by w1 equaling zero. If you remark out this instruction and recompile the program you will see that the window will momentarily flash open and then disappear. That's because without this loop the program continues execution to the next statement.
Note: Another way to do the exact same thing is:

Code:

```
WAITUNTIL IsWindowClosed(w1)
```

———————————————

Code:

```
END
```

Stops program execution when it is encountered.

———————————————

Newbie's at this point might ask, "where 's the call to the 'main' subroutine? It appears that it is not being used. And the 'main' subroutine has no parameters"

This is where some of the internal magic of IWBasic, and the OS, comes into play. As messages are generated and sent to an IWBasic window they are automatically passed to the handler routine. Because of that, there is no need for declared passed parameters in the handler SUB statement. By the same token, there is a returned value from the handler subroutine even though the SUB declaration doesn't specify one is required.

Let's look at each line of our simple handler routine above.

Code:

```
SUB main
```

The subroutine declaration for our handler routine.

———————————————

Code:

```
        IF @MESSAGE = @IDCLOSEWINDOW
```

This line is testing to see if the IWBasic system variable @MESSAGE contains a

value that is equal to the IWBasic system constant @IDCLOSEWINDOW. If the condition is true the next statement is executed; otherwise it is skipped.

Code:

```
            CLOSEWINDOW w1
```

This command closes the window and sets the window handle wHnd element of the WINDOW UDT for w1 to zero. When that occurs the WAITUNTIL loop discussed above will be true and the program will end.

Code:

```
        ENDIF
```

Marks the end of the IF block.

Code:

```
        RETURN 0
```

The standard return from the handler subroutine with the normal value of zero

---

Code:

```
ENDSUB
```

Marks the end of the handler subroutine.

There are a couple of points that can be made with this example. The first is that the reader should be able to see why I said earlier that the old DOS example was an example of a crude handler; specifically the example that only had the test for the 'Q' key being pressed (ex_2.iwb). In that example we where focusing on only one specific event and disregarding all other events. The same is happening with this simple windows example.

A second point is that we wrote no code associated with the button in the caption bar that we use to close the window. That code has to exist somewhere or the button would not cause that specific message to be sent. As our discussion continues that will be covered.

Also, we know that there are other ways to close the window; in task manager by stopping the process or my shutting down the computer which in turn closes our example program. The point being, as described earlier, that messages for a specific window can come from different places.

At this point, we need to explore the following line again.

Code:

```
        IF @MESSAGE = @IDCLOSEWINDOW
```

As previously stated, @MESSAGE is an IWBasic system variable which will contain a message ID. It is valid only inside a handler subroutine. In the past the IWBasic system variable @CLASS was used to hold the exact same information. They may be freely interchanged with each other.

Also, @IDCLOSEWINDOW is an IWBasic system constant. It's pre-assigned value is the same as the Windows OS message WM_CLOSE. They may be freely interchanged also. There are numerous OS message IDs that have had IWBasic system constants pre-assigned for them. In every case the user may substitute one for the other.
It must be noted that in order to use these non-IWBasic system constants the user will have to define them and assign the proper value. This step is not required if the user include's Sapero's 'windowssdk.inc' in the user's source file where that has already been taken care of.

---
the next section will continue the messages and handlers discussion

LarryMc

IonicWind Software

IWBasic => Create a Custom Control => Topic started by: LarryMc on July 30, 2011, 05:12:36 AM

Title: 4c. Messages and Message Handlers
Post by: LarryMc on July 30, 2011, 05:12:36 AM

Part C

To see a demonstration of messages being sent, let's modify our ex_4.iwb example:

ex_5.iwb
Code:
```
openconsole
DEF w1 as WINDOW
OPENWINDOW w1,0,0,350,350,@MINBOX|@MAXBOX|@SIZE,NULL,"Simple Window",&main
WAITUNTIL w1 = 0
closeconsole
END

SUB main
static int cnt=0
cnt++
? cnt,@MESSAGE
        IF @MESSAGE = @IDCLOSEWINDOW
                CLOSEWINDOW w1
        ENDIF
        RETURN 0
ENDSUB
```
Notice the three new lines added at the beginning of the 'main' handler:

Code:
```
static int cnt=0
cnt++
? cnt,@MESSAGE
```

The first line creates a STATIC, INT type, variable with an initial value of zero. The second line will increment that variable each time the subroutine is called. The third line will print (to the console window) the value of that variable as well as the message ID contained in @MESSAGE each pass through the handler subroutine.

Compile and run the program and perform the following actions while observing the contents of the console window.
1. Move the cursor around over the "Simple Window" client area (white portion).
2. Click/Double-click in the client area.
3. Move the cursor around over the caption area.
4. Click on the icon to the left of the title and then move the mouse over the resulting menu.
5. Drag the window by the caption bar.
6. Drag an edge of the window to resize the window.
7. Drag the caption bar of the console window so that a portion of that window covers a portion of the "Simple Window".

After playing with this for a few minutes two things should be obvious.
1. It doesn't take long for the cnt value to get rather large; meaning lots of messages are being sent.
2. Although the message numbers do change there appears to be a lot of

repetitions of the same numbers.

The latter would lead us to ask, "How can the same numbers appearing over and over be of any real value?" In order to address this question we need to go a little deeper into the discussion of messages.

Up until this point we've talked about messages as if they consisted of only a message ID. That was done to simply the discussion. Now is the time to look at the actual structure of messages. IWBasic has a command for sending messages and is defined in the IWBasic user's Guide as:

Quote

```
UINT = SENDMESSAGE(win as ANYTYPE, msg as UINT, wparam as UINT,lparam as ANYTYPE,OPT id=0 as UINT)
```

Let's examine each of the parameters:

win

Can be an IWBasic window, IWBasic dialog, or an OS window handle (wHnd).

———————————

msg

The message ID that has been the focus of discussion up to this point.

———————————

wparam

A single numeric value relative to the message being sent.

———————————

lparam

Any type of variable, constant or UDT relative to the message being sent.

———————————

id

Optional control id if message is being sent to a control.

———————————

Some messages have no associated data values resulting in wparam and lparam being set to zero. Other messages use either or both variables.

The user's Guide also has a comment that says this IWBasic command is a convenient replacement for the SendMessageA Windows API command. In reality the IWBasic command actually calls the API command. So what's the difference? The IWBasic version allows the user to pass a variable that is a window, dialog or hWnd and an optional control ID number. Remember I stated that the OS has no idea what a WINDOWS UDT type variable is. The OS system deals with handles. So, basically, all the IWBasic SENDMESSAGE command does it determine the proper hWnd value and plugs that into the OS API version.

NOTE: The key point to remember here is that as far as the OS is concerned, when it comes to sending messages, there is no difference between what we refer to as a window and what we refer to as a control.

So, the message ID is not the only piece of information being sent to a handler (via the @MESSAGE IWBasic system variable). The wparam value is sent via @WPARAM and the lparam value is sent via @LPARAM. As with @MESSAGE/@CLASS being interchangeable so are the @WPARAM/@CODE and @LPARAM/@QUAL pairs. The IWBaic User's Guide contains a section labeled 'Windows programming' which contains a sub-section labeled 'Message ID's' (under 'Messages and message loops') that contains numerous IWBasic system message constants and their windows OS equivalents. It is highly recommended that the entire section be read and studied further.

Back to our 'Simple Window' example (Ex_4.iwb). Let's look at the handler again.

Code:
```
SUB main
        IF @MESSAGE = @IDCLOSEWINDOW
                CLOSEWINDOW w1
        ENDIF
        RETURN 0
ENDSUB
```

Using an IF statement to check for each message we might want to use is rather inefficient. That is because each IF statement will be evaluated each pass through the handler routine. Considering that the handler routine can only be sent one message at a time there is a better way to do the testing. NOTE: Messages can be sent to a window at a faster rate than can be addressed at any given moment. A 'queue' is used for each window where messages are placed until they can be processed. This is all handled by the OS.

The following shows the restructured handler:

Code:
```
SUB main
        SELECT @MESSAGE
                case @IDCLOSEWINDOW
                        CLOSEWINDOW w1
        ENDSELECT
        RETURN 0
ENDSUB
```

Not a very impressive change but an important one. However, if we add all the messages that we might want to use, that already have IWBasic system variables defined, our example program will look something like this:

ex_6.iwb
Code:
```
openconsole
DEF w1 as WINDOW
OPENWINDOW w1,0,0,350,350,@MINBOX|@MAXBOX|@SIZE,NULL,"Simple Window",&main
WAITUNTIL w1 = 0
closeconsole
END

SUB main
        SELECT @MESSAGE
                CASE @IDLBUTTONDBLCLK
                CASE @IDLBUTTONUP
                CASE @IDRBUTTONUP
                CASE @IDMOUSEMOVE
                CASE @IDLBUTTONDN
                CASE @IDRBUTTONDN
                CASE @IDRBUTTONDBLCLK
                CASE @IDCONTROL
                CASE @IDMENUPICK
                CASE @IDMENUINIT
                CASE @IDMOVE
```

```
                CASE @IDMOVING
                CASE @IDSIZE
                CASE @IDSIZECHANGED
                CASE @IDSIZING
                CASE @IDHSCROLL
                CASE @IDVSCROLL
                CASE @IDINITDIALOG
                CASE @IDCANCEL
                CASE @IDCHAR
                CASE @IDKEYDOWN
                CASE @IDKEYUP
                CASE @IDTIMER
                CASE @IDPAINT
                CASE @IDCREATE
                        CENTERWINDOW w1
                CASE @IDBEFORENAV
                CASE @IDNAVCOMPLETE
                CASE @IDSTATUSTEXTUPDATE
                CASE @IDERASEBACKGROUND
                CASE @IDDESTROY
                CASE @IDCLOSEWINDOW
                        CLOSEWINDOW w1
        ENDSELECT
        RETURN 0
ENDSUB
```

If you compile and run this example you will notice something a little different happening. Even though we did not change the OPENWINDOW statement (which tells the OS to open the window in the upper left corner of the screen) the window opens in the middle of the screen. If you look at the list of CASE statements you will see @IDCREATE. That message is sent when a window is going to be opened. So the window is created in the upper, left corner of the screen, relocated, and then displayed, in that order.

The reader might say, "why put the CENTERWINDOW command there? Why not just put it on the line following the OPENWINDOW command?" That's a valid question. The answer is that it is a matter of sequence. By placing the command at that location the window will be created in the upper, left corner of the screen, displayed, and then relocated, in that order. This will result in the window momentarily flickering in the corner before appearing in the center of the screen. For the same reason the @IDCREATE message handler is a good place to set fonts, colors, and to initiate data variable displays. Likewise, the @IDCLOSEWINDOW message can be a good place to put code that will save data before the window is closed

All the messages listed in the example are touched upon in various places in the IWBasic User's Guide. There are two points to made here, again. One is that we can pick and choose which messages benefit us and we will use. The other is that messages we choose not to use are still making things happen. A good example is @IDLBUTTONUP, which is the left mouse button being released. That will cause our window to be displayed on top of other windows if it is not already. So something is taking care of these things somewhere.

At this point we need to take a look at controls. After our review there, we will return here and continue.

IonicWind Software

IWBasic => Create a Custom Control => Topic started by: LarryMc on August 08, 2011, 11:46:05 AM

Title: 5a. IWBasic Controls
Post by: LarryMc on August 08, 2011, 11:46:05 AM

Part A

As was said about opening a window in IWBasic being extremely easy, so is adding a control to a window. We will use the same methodology to explore a control. Controls are created with the following command line:

Quote

```
CONTROL parent, type, title, left, top, width, height, style_flags, id
```

This is simply a call to a subroutine that is sent nine parameters (pieces of information) to use to do whatever the subroutine has to do to open our control. Let's look at those parameters.

parent

Unlike a window, all controls must have a parent window. This parameter will be the WINDOWS type UDT variable name of the control's parent.

type

An IWBasic predefined constant variable identifying the type of control to create. The available IWBasic control types are:
@BUTTON
@CHECKBOX
@RADIOBUTTON
@EDIT
@LISTBOX
@COMBOBOX
@STATIC
@SCROLLBAR
@GROUPBOX
@RICHEDIT
@LISTVIEW
@STATUS
@SYSBUTTON
@RGNBUTTON
@TREEVIEW

title

The text of the control

left, top, width, height

The location and size of the control in pixels. Where the location parameters of a window were relative to the screen, the location of a control is relative to the upper left corner of the parent windows client area.

style-flags

This is a single UINT whose individual bits each have some specific impact on the appearance or functionality of the control. Usually, to make things easier to read, each bit is given a CONST definition representing one bit and various bits are or'd together with the' |' operator.

id

A unique UNIT type variable, constant, or literal number that identifies the specific control.

_____

Notably absent in the list of passed parameters is a handler variable. Controls both send and receive messages. All messages that are sent by controls are automatically routed to their parent's message handler by the OS.

Let's modify example ex_6.iwb to show how to add a simple button to our window and see what message we can generate.

Ex_7.iwb

Code:

```
openconsole
DEF w1 as WINDOW
OPENWINDOW w1,0,0,350,350,@MINBOX|@MAXBOX|@SIZE,NULL,"Simple Window",&main
CONTROL w1,@button,"1st",40,40,50,20,0,1
CONTROL w1,@button,"2nd",140,40,50,20,0,2
WAITUNTIL w1 = 0
closeconsole

SUB main
        SELECT @MESSAGE
                CASE @IDCONTROL
                        ? "control message"
                CASE @IDCLOSEWINDOW
                        CLOSEWINDOW w1
        ENDSELECT
        RETURN 0
ENDSUB
END
```

In ex_7.iwb we have added two buttons immediately after the OPENWINDOW command. In the windows handler all but two predefined messages have been removed. In the CASE @IDCONTROL portion of the handler there is a print statement.

Compile and run the example. Make sure the console window is not covered by the "Simple Window". Click on various locations in the client area of window including on the two buttons. Notice that when either of the buttons are clicked that the "control message" text is printed in the console window. This indicates that each time an event occurs involving a control that an @IDCONTROL message is sent to its parent window.

But this isn't telling us which control has had an event. IWBasic provides us that information. When an @IDCONTROL is received IWBasic loads a system variable with the identify of the control sending the message. This system variable is @CONTROLID. We can modify our example so that this can be seen.

Ex_8.iwb

Code:

```
openconsole
DEF w1 as WINDOW
OPENWINDOW w1,0,0,350,350,@MINBOX|@MAXBOX|@SIZE,NULL,"Simple Window",&main
CONTROL w1,@button,"1st",40,40,50,20,0,1
```

```
CONTROL w1,@button,"2nd",140,40,50,20,0,2
WAITUNTIL w1 = 0
closeconsole
END

SUB main
        SELECT @MESSAGE
                CASE @IDCONTROL
                        ? @CONTROLID
                CASE @IDCLOSEWINDOW
                        CLOSEWINDOW w1
        ENDSELECT
        RETURN 0
ENDSUB
```

When this example is ran the ID used when the control was created is printed when the control is clicked. But let's be a little more precise in our test. Press the left mouse button over one of the buttons and hold it down. Notice nothing is printed. Let the button up. Notice that the buttons ID is printed. Right click on a button. Nothing happens. So, the button only sends a message when the left mouse button is released. This specific operation has been programmed into the button. We know that there is a left button pressed event because we see the appearance of the button change when we press the button. So, somewhere there was a decision made to have the button change its appearance when the button was pressed and then again when it was release but only for the left mouse button. And somewhere the code to do this has to reside.

Before we go further, let's take another look at where the values for these system variables are coming from. In the Messages and Message Handlers - Part C section we learned that all messages are ultimately sent with the SendMessage API command. Also, we learned that when a window handler receives a message, IWBasic loads @MESSAGE, @WPARAM and @LPARAM with the message's information.

In our example we know that @MESSAGE contains a value equal to @IDCONTROL, which tells us a control is send a message. Let's modify our example to print @WPARAM and @LPARAM.

In ex_8.iwb change this line:

Code:

```
                        ? @CONTROLID
```

to

Code:

```
                        ? @CONTROLID, @WPARAM, @LPARAM
```

Compile and run. Click on the two buttons and observe the results in the console window.

We see that @WPARAM contains the clicked control's ID from when it was created. @LPARAM contains a large number. The value appears to be constant for a specific control.

In the window discussion we stated that the OS doesn't know anything about IWBasic variable names and such. The OS assigns handles to windows when they are created. The number contained in @LPARAM is the control's handle.
We can prove this very easily. IWB contains a function, GETCONTROLHANDLE, where the parent window and control ID variables are passed in and the OS handle is returned.

In ex_8.iwb change this line:

Code:

```
                        ? @CONTROLID
```

to

Code:

```
                    ? @CONTROLID, @WPARAM, @LPARAM, GETCONTROLHANDLE(w1,@CONTROLID)
```

Compile and run. Click on the two buttons and observe the results in the console window. In each case the first two numbers are the same (control's ID) and the last two numbers are the same (the control's OS handle).

So far, our testing of controls has involved only buttons. We really need to try it with another type control to see if our "rule" is correct. We'll use a simple edit control for this purpose.

Taking our last, modified version of ex_8.iwb and adding the edit control results in:

Ex_9.iwb

Code:

```
openconsole
DEF w1 as WINDOW
OPENWINDOW w1,0,0,350,350,@MINBOX|@MAXBOX|@SIZE,NULL,"Simple Window",&main
CONTROL w1,@button,"1st",40,40,50,20,0,1
CONTROL w1,@button,"2nd",140,40,50,20,0,2
CONTROL w1,@edit,"3rd",240,40,50,20,0,3
WAITUNTIL w1 = 0
closeconsole
END

SUB main
        SELECT @MESSAGE
                CASE @IDCONTROL
                        ? @CONTROLID,@WPARAM,@LPARAM,GETCONTROLHANDLE(w1,@CONTROLID)
                CASE @IDCLOSEWINDOW
                        CLOSEWINDOW w1
        ENDSELECT
        RETURN 0
ENDSUB
```

Compile and run. Right off the bat you notice a difference. Two lines are printed before you have had a chance to do anything. As before with the buttons the last two numbers match on each line (the control's OS handle). But notice that the first two numbers don't match. However, the first number is the id we assigned to the edit control when it was created so that part is correct. We had said earlier that the @CONTROLID variable was set to the value of the @WPARAM when it was sent in the message via the wparam argument. It appears something else is going on here that wasn't seen with the button controls. For our "rule" to be true the control's ID has to be contained in the second number(@WPARAM).

In order to reduce the clutter let's change this line:

Code:

```
                    ? @CONTROLID,@WPARAM,@LPARAM,GETCONTROLHANDLE(w1,@CONTROLID)
```

to

Code:

```
                    ? @CONTROLID,@WPARAM
```

In the declaration statement for the SENDMESSAGE command it states that the wparam parameter is a UNIT which means it is a 32 bit number.

Let's look at the lower 16 bits of @WPARAM. We can accomplish this by ANDing the variable with 0xFFFF. To see the result, change

Code:

```
                    ? @CONTROLID,@WPARAM
```

to

Code:

```
                    ? @CONTROLID,@WPARAM & 0xFFFF, @WPARAM
```

Compile and run. Observe that the second number now properly reflects the

control's IWB ID. So, our previous "rule" that the ID was contained in @WPARAM is true but we need to be more precise and say it is contained in the lower 16 bits of that variable.

We can just throw away the high order bits. They have to be telling us something. Let's discard the 16 low order bits and shift the high order bits down 16 bits. This will give us the 16 high order bits as a 0 based number. We accomplish this by simply dividing the variable by 0xFFFF.

Replace this line:
Code:
```
? @CONTROLID,@WPARAM & 0xFFFF, @WPARAM
```
with
Code:
```
? @CONTROLID,@WPARAM & 0xFFFF, @WPARAM / 0xFFFF
```

Compile and run. Notice the pattern of numbers. When we run the example we get to lines (messages) without any activity from the User.
If we click in the edit control we get a third number but only the first time it is clicked in a sequence. We get a fourth number when we click outside the edit control but, again, only the first time in a sequence. And finally, each time we change the contents of the edit control we get the two message sequence we got originally.

The following example shows what those changing numbers mean.
Ex_10.iwb
Code:
```
openconsole
DEF w1 as WINDOW
OPENWINDOW w1,0,0,350,350,@MINBOX|@MAXBOX|@SIZE,NULL,"Simple Window",&main
CONTROL w1,@button,"1st",40,40,50,20,0,1
CONTROL w1,@button,"2nd",140,40,50,20,0,2
CONTROL w1,@edit,"3rd",240,40,50,20,0,3
WAITUNTIL w1 = 0
closeconsole
END

SUB main
        SELECT @MESSAGE
                CASE @IDCONTROL
                        ? "@ENUPDATE= ",@ENUPDATE
                        ? "@ENCHANGE= ",@ENCHANGE
                        ? "@ENSETFOCUS= ",@ENSETFOCUS
                        ? "@ENKILLFOCUS= ",@ENKILLFOCUS
                        ? @CONTROLID,@WPARAM & 0xFFFF,@WPARAM / 0xFFFF
                CASE @IDCLOSEWINDOW
                        CLOSEWINDOW w1
        ENDSELECT
        RETURN 0
ENDSUB
```
This example has four extra print statements that print the values of four IWB system constants. As with many other constants their names and values are based upon OS constants. These constants, and more, for edit controls can be found in the IWB User's Guide under *Windows Programming / Controls / Edit Controls*.

Compile and run. Perform the various activities and observe the nature of the messages.

Now, notice that each of the message variables start with @EN. The E is for edit controls and the N stands for notifications. Just a windows receive hundreds of messages directly so do controls. Controls report many of their messages to their parent window to allow for responses. The messages are called notification messages. They are still sent with the same SENDMESSAGE type message

structure.

IWB provides us an easy way to obtain these notification message ids by doing what IWB does for us with the other message variables we've covered.
When IWB decodes the @WPARAM to put the control ID into @CONTROLID it puts the notification message ID in the IWB system constant @NOTIFYCODE.
Change this line:
Code:

```
? @CONTROLID,@WPARAM & 0xFFFF,@WPARAM / 0xFFFF
```

to
Code:

```
? @CONTROLID,@WPARAM / 0xFFFF,@NOTIFYCODE
```

Compile and run. This confirms that @NOTIFYCODE contains the data from the high order bits of @WPARAM.

Not all controls send notification messages. Click one of the buttons. The notification message ID is 0. That is because the sending of the message is enough by itself without any additional clarification (notification) information as to the type of event that occurred.

IonicWind Software

IWBasic => Create a Custom Control => Topic started by: LarryMc on August 10, 2011, 08:49:45 AM

Title: 5b. IWBasic Controls
Post by: LarryMc on August 10, 2011, 08:49:45 AM

Part B

IWBasic provides us with another command for creating controls, CONTROLEX. Controls are created with the following command line:

Code:

```
CONTROLEX parent, class, title, left, top, width, height, style_flags, exStyle_flags, id
```

parent, title, left, top, width, height, style_flags, id

These parameters are exactly the same as described in the CONTROL command discussion above.

exStyle_flags

These flags work exactly the same as the *style_flags* discussed previously. The difference is that, while *style_flags* are control type specific, *exStyle_flags* are more global in nature. These flags control such things as visibility and the type of border the control has.

class

This is a string that identifies the specific type of control being added. To that extent it is doing what the *type* parameter does in the CONTROL command.

What the reader needs to understand is the control *type* constants are provided as a convenience to the user so that a *class* string doesn't have to be entered for each control. That implies that the OS requires the *class* string.

_____

Coming Next: Pulling It Together

IonicWind Software

IWBasic => Create a Custom Control => Topic started by: LarryMc on August 12, 2011, 07:54:54 PM

Title: 6. Pulling It Together
Post by: LarryMc on August 12, 2011, 07:54:54 PM

In order to get a better understanding of how everything fits together we need to look "under the hood", so to speak, and see what is going on that we don't normally see.

The Windows API is a collection of functions, provided by the OS, that allows us to have a graphic interface to our programs. These functions are often cryptic and can be extremely complex for the casual programmer.

As previously stated, IWBasic commands, in general, make our programming life simpler. They tend to isolate us from the API. The IWB command may be very close in syntax to the API function. The SENDMESSAGE command is a good example. It is simply a 'wrapper' that takes IWB type variables and converts them to the proper format before calling the API SendMessage function. Simple and straightforward. Some IWB commands have the same name as an API function. These naming conflicts are commonly resolved via an "alias" declaration for the API function and the prepending of an underscore '_' to the function name. That is why you will see those underscored functions scattered through the sample files.

Other IWB commands invoke several API functions while performing its designated function. Often, the IWB command is calling functions that we are totally unaware of.

We have already discussed three IWB commands that are of this more complex nature:
- OPENWINDOW
- CONTROL
- CONTROLEX

The most important API function called by all three of these IWBasic commands is CreateWindowExA. This function is the one that actually creates the object (window/control).

In the previous discussion about messages we learned that to the operating system it didn't matter whether a window or control was sending/receiving messages. To the operating system they were the same. We now know why. They are created the same. That leads us to the old adage that "if it walks like a duck and quacks like a duck then it must be a duck."

Let's examine the CreateWindowExA API function in relation to the three IWBasic commands above:

Code:
```
CreateWindowExA(exStyle_flags, class, title, style_flags, left, top, width, height, parent, hMenu, hInstance, lpParam),INT
OPENWINDOW variable, left, top, width, height, style_flags, parent, title, handler
CONTROL parent, type, title, left, top, width, height, style_flags, id
CONTROLEX parent, class, title, left, top, width, height, style_flags, exStyle_flags, id
```

We'll start by removing parameters that are common to all four entries and that have been previously discussed. They are:
- *parent*
- *left*, *top*, *width*, *height*
- *title*

That leaves us with following parameters for each:
Code:
```
CreateWindowExA(exStyle_flags, class, hMenu, hInstance, lpParam),UINT
OPENWINDOW variable, handler
CONTROL  type, id
CONTROLEX class, exStyle_flags, id
```

The CreateWindowExA returns a window handle when it called successfully. We know from previous discussions that that handle is stored in the variable UDT element 'hWnd' when using OPENWINDOW. We also know that when dealing with controls the handle is tied to the control's id and shows up in the @LPARAM of a message.

The CreateWindowExA parameters *hMenu* and *lpParam* are used internally by IWBasic and the OS and have different purposes depending upon whether we are dealing with what we call a dialog or window; or a regular window vs. a MDI window; or a control. It is beyond the scope of this tutorial to go into those details.

hInstance is the handle to the current module/application.

Removing all of those we have:

Code:
```
CreateWindowExA(exStyle_flags, class)
OPENWINDOW  handler
CONTROL  type
CONTROLEX class, exStyle_flags
```

Suffice it to say that the OPENWINDOW and CONTROL commands automatically assign some *exStyle_flags* based upon what type of object we are dealing with. Setting those aside we are left with:

Code:
```
CreateWindowExA(class)
OPENWINDOW  handler
CONTROL  type
CONTROLEX class
```

CreateWindowExA requires a *class* parameter in order to create an object, be it a window or control. From the section on controls we know that the *class* parameter is "a string that identifies the specific type of control being added". Since it has to apply to both windows and controls we can modify that definition to "a string that identifies the specific type of object being added".

Looking at the source code for OPENWINDOW we find that the command is assigning the *class* for us.
For what we refer to as a normal IWBasic window the *class* is "IWBasicWndClass". When creating a window with the @MDIFRAME style the frame is assigned the "IWBasicFrameClass" *class*. At the same time the associated client window is assigned the "MDICLIENT" *class*.

The CONTROL command has a SELECT/ENDSELECT block that looks at the *type* constant passed to the command. The following lists each predefined *type* and the corresponding automatically assigned *class*:

IWBasic TYPE Constant Class
@RGNBUTTON "BUTTON"
@BUTTON "BUTTON"
@SYSBUTTON "BUTTON"
@CHECKBOX "BUTTON"
@RADIOBUTTON "BUTTON"
@GROUPBOX "BUTTON"
@EDIT "EDIT"
@LISTBOX "LISTBOX"
@COMBOBOX "COMBOBOX"
@STATIC "STATIC"
@SCROLLBAR "SCROLLBAR"
@RICHEDIT "RichEdit"
@LISTVIEW "SysListView32"
@STATUS "msctls_statusbar32"
@TREEVIEW "SysTreeView32"

Notice that the first six *types* are all of the same *class*. What makes them appear different when they are created is *style_flags*. The CONTROL command automatically assigns the proper *style_flag* for the desired control.

The is a good time to note that the OPENWINDOW and CONTROL commands also call API functions to automatically set default fonts, foreground/background colors as well as appropriate *exstyle_flags*.

Well, that's all fine and good, but what is the *class* doing for us? Why do we need to be concerned about it?

Remember back in the sections on messages and message handlers. We said that the OS sorts out what messages get sent where. Well, when we create an object we tell the OS which handler to use for that object and what messages to send to the handler.

But wait you say, the CreateWindowExA, CONTROL, and CONTROLEX don't have a message handler variable. Only the OPENWINDOW command has anything identifying a message handler. I know it may seem strange when I say that the OPENWINDOW command is the special case here and the CONTROL and CONTROLEX commands are the more normal cases.

Each *class* has a dedicated message handler tied to it. The *class* name is simply a way to identify which handler to use.
But wait, we define a message handler when we use OPENWINDOW. Something doesn't sound right. The window handler defined with the OPENWINDOW is there so that it is easy for us to integrate our code with the real windows handler defined by IWBasic that we can't see. This setup makes it easier on us to do our thing,

code wise. That is what allows us to pick our own name for a handler routine.

Fine, but what about message handlers for individual controls. We don't define them anywhere.

But they are. When a control is created (in this case we are talking about when it is designed initially) it has a message handler that is an integral part of the control. It totally defines the control; what it looks like; what it responds to; everything. The creator of the control has to give the control a unique *class* name to identify it, in addition, it requires a unique handler name which the ultimate user of the control will never see.

The one missing piece we haven't touched on is the RegisterClassExA API function. For every *class* this function has to be called to provide the OS the necessary information to functionally merge the object into the system. Although your application may contains hundreds of objects of a given *class* (or type) the *class* has to be registered only once. The function passes a UDT type OS structure with, among other things, the *class* name and a pointer to the default message handler subroutine. We will cover the structure in more detail when we start developing the code for our custom control.

For the standard controls the registering of the *class* names and the default handlers is taken care of by the system. However, IWBasic does sub-class some of the handlers in order to change certain aspects of the regular OS controls. Sub-classing is not within the scope of this tutorial.

To conclude our review we summarize as follows:

When a GUI application is built we create windows and controls that generate and/or respond to event messages. Part of what happens when events occur is determined by the OS system and part can be determined by the user. Some events are responded to in a manner predefined by IWBasic. We can create our own windows and/or controls that act totally the way we desire by creating a new class and handler and registering that class with the system. But, when we choose to do that, we loose a lot of the ease of coding that IWBasic normally provides us. However, we are ultimately providing new functionality for user's of our creations along with the same normal ease of use that IWBasic normally provides.

Hopefully, at this point, the reader has a better understanding of how IWBasic and the OS are both dealing with windows/controls.
I highly suggest that the Windows Programming section of IWBasic's User's Guide be reread before proceeding, to reinforce what has been written in this Review section.

_____

Coming Next - User's Control Spec

LarryMc

IonicWind Software

IWBasic => Create a Custom Control => Topic started by: LarryMc on August 13, 2011, 07:43:17 PM

Title: 7. User's Control Spec
Post by: LarryMc on August 13, 2011, 07:43:17 PM

The first step in building a custom control is defining what you want the control to do. On the surface that sounds like a simple, no-brainer. We could say we want to build a grid control or a chart control and end it there.

If that is the extent of your initial definition and planning you'll most likely create headaches for yourself as you get deeper into the coding. You may find you have coded yourself into a hole that you can only get out of by discarding days, weeks, or even months of coding (depending upon the complexity of your control). Extra time spent at this stage will pay dividends in the long run.

What I chose for my first attempt at creating a custom control (and the basis for this tutorial) was a round gage. I spent almost 11 years in the Air Force with a great deal of my time spent in the cockpit of numerous kinds of aircraft. After that I spent 25 years in a chemical plant. As a result, when I started the gage control project I had a pretty good idea of what the final product should look like and how it should work.

For the sake of this tutorial let's define all that we can about where we want to wind up, starting with appearance:
- a round gage with a black border/frame
- a white or gray dial face
- circular scale on dial face with evenly spaced numbers and tick marks
- multiple dial face layouts/types

The various dial face layouts/types

CLOCK (Fig. 7, Fig. 8)
- 24 hr/12 hr scale
- 1 minute tick marks on 12hr scale
- 30 minute tick marks on 24hr scale
- black second hand
- long red minute hand
- short red hour hand
- text label above center of gage

270 DEGREE (Fig.1, Fig. 2, Fig. 3, Fig.4)
- scale covers 270 degrees (similar to a car's speedometer)
- four styles determined by 90 degree unused area being at bottom, top, left, or right
- 1 to 15 major tic marks w/ numbers
- one minor tic mark between major tic marks
- single red pointer
- text identifier above center of gage
- range multiplier/units below center of gage

360 DEGREE - SINGLE (Fig. 5)
- full circle scale

- fixed scale of 0-9
- intermediate tic marks
- odometer type counter below gage center
- single red pointer
- 1 rev of pointer inc/dec counter by one
- text identifier above center of gage
- range multiplier/units below center of gage

360 DEGREE - DOUBLE (Fig. 6)
- full circle scale
- fixed scale of 0-9
- intermediate tic marks
- odometer type counter below gage center
- short red pointer
- long red pointer
- 1 rev of short pointer inc/dec counter by one
- 10 revs of long pointer = 1 rev of short pointer
- text identifier above center of gage
- range multiplier/units below center of gage

2-PEN VERTICAL (Fig. 9)
- 2 vertical scale arcs in center of gage that share scale numbers
- numbers fixed @ 0-10 in vertical column w/tic marks
- 2 independent red pointers with pivot points at left and right edges of gage
- text identifiers above center of gage on left/right for each pointer
- range multipliers/units below center of gage on left/right for each pointer

2-PEN HORIZONTAL (Fig. 10)
- 2 horizontal scale arcs in center of gage that share scale numbers
- numbers fixed @ 0-10 in horizontal row w/tic marks
- 2 independent red pointers with pivot points at top and bottom edges of gage
- text identifiers and range multipliers/units above center of gage for top pointer
- text identifiers and range multipliers/units below center of gage for bottom pointer

Next we need to identify the things we need to be able to configure/control:
- set the style/type of gage
- set the overall size of the gage
- set the max min values for the raw data sent to a gage
- set the max min values for the displayed range on a gage
- set the text identifier for a gage/pointers
- set the range multiplier/units text for a gage/pointers, if appropriate
- set the offset of clocks to show time zones other than local time.
- set the color of rectangle containing the gage
- set the white/grey color of the dial face
- set the current position of pointer(s), where appropriate

_____

Coming Next - Dynamic Link or Static Library

IonicWind Software

IWBasic => Create a Custom Control => Topic started by: LarryMc on August 14, 2011, 04:17:45 PM

Title: 8. Dynamic Link or Static Library
Post by: LarryMc on August 14, 2011, 04:17:45 PM

When you are finished with your custom control creation you will more than likely want to share it with others. There are basically three ways to package your creation for sharing. Note: This discussion has no bearing in whether you sell or give it away.

First, you can share the raw source code. That is ultimately what is being done with this tutorial. But then again, it would be rather difficult to do a tutorial on creating a custom control without showing the code. However, I have a hidden agenda. I'm hoping this tutorial will encourage one or more brave programmer's to create something unique and/or useful.

Normally, a custom control is shared in the form of a library. Okay; what's a library. A library is simply a collection of compiled source file subroutines. It may also contain some variables that are global to the code in the library. During development, the source code that will eventually be in the library is no different then the code you would write for any other program you create in IWBasic.

You decide to go with the norm and share your control with a library. But you're not done with making this decision. Are you going to use a dynamic link library (DLL) or a static library (LIB).

Both types of libraries contain your compiled source code. IWBasic is capable of creating either type. The big difference is how they are used by the people you share with.

A static library is used only by a programmer when they are creating an application. When they compile their application the object modules inside your library are copied (linked) into their application. In turn, the user of their application never really knows your static library exists. You already use static libraries every time you compile a project in IWBasic without even knowing it.

With a dynamic link library, the programmer has to first create a linking library from the dynamic library. This linking library doesn't contain any of your original compiled object files. It only contains information that identifies your DLL and the addresses of all the subroutines/functions in your DLL that are available for the programmer to use in creating their application. When the programmer distributes their application they have to also distribute your DLL along with it. That is because your object code is only loaded into memory to be executed at run time. You also use DLL's every time you compile a program now that uses a IWBasic command that in turn uses one of the functions in the OS's DLLs.

To me, a static lib is the simplest and cleanest to use; for you while you are creating it, for the programmer's you share it with, and for the user's of their applications that use your library.

In this tutorial we will be creating our custom control and sharing it as a static library.

Coming Next - Development Environment

IonicWind Software

IWBasic => Create a Custom Control => Topic started by: LarryMc on August 16, 2011, 12:25:26 PM

Title: 9. Development Environment
Post by: LarryMc on August 16, 2011, 12:25:26 PM

It's obvious that we going to develop our custom control in IWBasic's IDE. So, this short discussion is more about the details of how we are going to go about that.

We previously stated that we've going to share our custom control as a static library. And we know that the user will have to have a '$USE "OurLib.lib" statement somewhere in there source code in order to be able to use our custom control. It would seem to follow that we would need the same sort of arrangement while we are testing our custom control as we develop it.

The setup process would flow along these lines:
- We create an IWBasic project, with the Project Type as 'Static Library'.
- We create an IWBasic source file to contain our actual custom control code.
Note: We must acknowledge that there is absolutely no reason why we can't use more than one source file for our control.
- We add our source files to the project.
- We create a separate source file as a 'Windows' type exe that we will use for testing our library.
- We create an include file that has the available functions we can call from our library, along with any supporting constants..
- We add the include file to our testing file.
- In either the include file or the testing file we add our '$USE' directive.

With the above setup, a normal editing session would go something like this:
- Open our control project.
- Make some changes.
- Compile the project to create the .lib file.
- Close the project.
- Open our testing program.
- Make any changes in order to do the appropriate testing.
- Compile and run our testing program.
- Reopen our project to make the next set of changes to the control and repeat this process.

As can be seen from this sequence, for each set of changes during development, we have to open and close the library project and compile both the library project and our testing file. Not very efficient.

This is the setup process I chose to use:
- Create an IWBasic project, with the Project Type as 'Windows EXE'.
- Create an IWBasic source file to contain our actual custom control code.
- Create an IWBasic source file that contains our testing code with the $MAIN directive at the top.
- Add both files to our project.

With this arrangement we don't have to keep opening and closing a project between edits and we only have to perform one compile per set of edits.

After we have completed the development then we can separate the testing file from the custom control file. The control portion can then be added to a new project in order to create the lib file. The testing file can possibly be modified to become an example file (with the addition of the proper include file and directives).

Before this tutorial is finished the reader will have examples of all phases of this total process.

_____

Coming Next - Control Required Components

IonicWind Software

IWBasic => Create a Custom Control => Topic started by: LarryMc on August 16, 2011, 08:34:12 PM

Title: 10. Control Required Components
Post by: LarryMc on August 16, 2011, 08:34:12 PM

Here we are on the verge of actually writing some code for our custom control. Now is a good time for us to discuss what the required components for a custom control are. Notice that I said "a" custom control and not "our" custom control. That is because this tutorial covers almost all the required components of any custom control. The reason I say "almost" is because of the particular custom control we will create in this tutorial. It doesn't need a couple of the required components.

It should be obvious, after the amount of time spent talking about message handlers, that we will need our own custom message handler. Message handlers respond to messages. We know, from the previous discussions, that the OS sends a lot of messages and we can pick and choose what messages we will write code for. We will need to handle the @IDPAINT / WM_PAINT message since that is where we will actually draw the control. As we proceed we will also address a couple of other OS messages we will need to use.

We will also need some user defined messages so that the user's application can communicate with our control to configure it and to update the pointers.

We could just use the SENDMESSAGE command to send the messages (with our information) to our control, but instead, we will create commands the user can use.

Since our custom control message handler will have to handle an unlimited number of our controls in any given user application we have to have a way to store the configuration/runtime information that is unique for each instance of our control. The OS gives us several methods of doing this. We will use the method of my choosing and briefly mention the others that I discovered.

The OS won't send any messages to our handler until we register it as a class with the OS. So we will have to address that.

Can't forget that we have to have a way for the user to create an instance of our control. We could structure things so the CONTROLEX command could be used. However, I chose to create a custom command like was done in IWBasic's ControlPak.

When we create the lib that we will share, we will also need to create an include file (*.inc) for the user that contains all the required declarations to support our control.

And finally, we need to create some sort of help discussion to given the user the necessary information in order to properly implement our control.

With that said, we have finally arrived at the point of writing some code.

_____

Coming Next - Creating a Skeleton Project

IonicWind Software

IWBasic => Create a Custom Control => Topic started by: LarryMc on August 17, 2011, 08:05:03 PM

Title: 11. Creating a Skeleton Project
Post by: LarryMc on August 17, 2011, 08:05:03 PM

We've finally arrived at the point of actually writing code for our custom control. We'll start by creating an IWBasic project to hold both our testing code and our control code.

Select *File/New/Project* from the IWBasic IDE main menu. This will open a *New Project* dialog. Make the following entries:
- Project Name -'CCT'
- Project Path - A path of your choosing.
- Project Type - 'Windows Executable'
- Output File - 'CCT.exe'

and then click *OK*. Leave the project open.

Now we need to create our two source files.

Select *File/New/IWBasic Source File* from the IWBasic IDE main menu. This will open an empty IWBasic source file with an automatically generated name of IWBASICx where x is a sequential number.

Select *File/SaveAs* from the IWBasic IDE main menu. This will open a *Save As* dialog. Navigate to the folder specified in the Project Path entry above. Enter 'CCT_test.iwb' for the file name and click *Save*. Right click on the just renamed file and select *Insert File Into Project*. The popup menu will disappear and the file name will appear in the IDE's *File View* panel in the lower right corner of the IDE.

Create the second source file following the exact same procedure, saving the file as 'CCT_lib.iwb' and insert it into the project.

CCT_test.iwb

The CCT_test file will contain code that we need to test our control code as we develop it. Once we are through with development we will convert the file to an example.
The following shows the code in the different parts of the skeleton.

Code:
```
/*============================================================
*       CCT_test.iwb
*       Custom Control Tutorial Testing File
*  PART 01
*       Copyright (c)T.L. McCaughn 2011
*  This file may be freely shared
*============================================================*/
```

I have a habit of putting title blocks on my source files. The significant thing here is that as we progress through the development of our control I will be creating updated versions of this file. This one is identified as "Part 1".

Next is:

Code:
```
        AUTODEFINE "OFF"        /* I always use this - personal preference */
        $MAIN                   /* Reguired for this develpoment project not required when making single example file */
        $include "windowssdk.inc"  /* I always use this - personal preference */
```

AUTODEFINE "OFF" - keeps me from inadvertently assigning the wrong type of value to a variable.
$INCLUDE "windowssdk.inc" - saves me a lot of time in not having to look up declarations for system functions, except in rare cases.
$MAIN - All projects require a $MAIN directive. It is used to define where program execution starts when there are multiple source files. When this source file is converted to a single file example this directive can be removed.

Code:
```
/*----------------SECTION 1---------------------------------
 This section contains declarations that will be converted into an include file that
 User's will use in there applications with the following line:
$INCLUDE "myControl.inc"
------------------------------------------------------------*/
```

```
        '$USE "myControl.lib    /* uncomment when converted to include file */

/*=============== END OF SECTION 1 ==========================*/
```

Section 1 contains constant definitions and function declarations that are required for the test program to interact with the custom control code. When the test file is converted to an example the contents of this section will be replaced by an include file with the $INCLUDE "myControl.inc" directive. The include file will contain all the code previously in this section. Also, at that time, the $USE "myControl.lib directive will be uncommented. This include file will allow user's to link the custom control library into their applications.

Code:

```
/*----------------SECTION 2--------------------------------
 This section contains declarations needed for the test program
and the converted example along with any needed initialization
-------------------------------------------------------------*/
        window  main
        int     run
/*=============== END OF SECTION 2 ===========================*/
```

Section 2 contains constant definitions and function definitions that are required by the test/example program itself. There is no direct connect between them and the library other than as interim variables that are passed to the control.

Code:

```
/*----------------SECTION 3--------------------------------
 This section creates our test/example parent window and its associated parts
-------------------------------------------------------------*/
        openwindow main, 0, 0, 880, 780, @CAPTION|@SYSMENU, 0, "Gage Demo", &mainHandler
        centerwindow main
        BEGINMENU main
                MENUTITLE "&File"
                        MENUITEM "Quit", 0, 100
        ENDMENU

/*=============== END OF SECTION 3 ===========================*/
```

Section 3 contains the basic creation and setup code for any standard window.

Code:

```
/*----------------SECTION 4--------------------------------
 This section is where we create one or more instances of our control.
-------------------------------------------------------------*/


/*=============== END OF SECTION 4 ===========================*/
```

Section 4 is where we place our user code that creates an instance of our control.

Code:

```
/*----------------SECTION 5--------------------------------
 This section is where we configure each instance of our control.
-------------------------------------------------------------*/


/*=============== END OF SECTION 5 ===========================*/
```

After creating our controls in Section 4 we will configure them in Section 5.

Code:

```
/*----------------SECTION 6--------------------------------
 This section sets the timer which will update our controls.
-------------------------------------------------------------*/
        STARTTIMER main, 100, 42
        run = 1
/*=============== END OF SECTION 6 ===========================*/
```

Section 6 is where we create a timer to be use to trigger updates to our controls. The shown code indicate timer # 42 will be set to 1 tenth of a second intervals.
We are also setting up our idle test by setting run to 1.

Code:

```
/*----------------SECTION 7--------------------------------
 This section is the idle loop for our test/example program.
-------------------------------------------------------------*/
        WAITUNTIL run = 0
/*=============== END OF SECTION 7 ===========================*/
```

This is our idle loop where messages are being dispatched in the background by the OS.

Code:

```
/*---------------SECTION 8-----------------------------------------
 This section is where we do any cleanup and close our test/example program
-------------------------------------------------------------------------*/
        STOPTIMER main, 42
        CLOSEWINDOW main
        END

/*=============== END OF SECTION 8 ==========================*/
```

In Section 8, we do our stand cleanup before shutdown once a closewindow event has occurred.

Code:
```
/*---------------SECTION 9-----------------------------------------
 This section contains the message handler for our test/example program..
-------------------------------------------------------------------------*/
SUB mainHandler(),INT
   SELECT @MESSAGE
        CASE @IDCLOSEWINDOW
                run = 0
       CASE @IDTIMER
                SELECT @CODE
                        CASE 42
                ENDSELECT
        CASE @IDMENUPICK
                SELECT @MENUNUM
                        CASE 100
                                run = 0
                ENDSELECT
   ENDSELECT
   RETURN 0
ENDSUB
/*=============== END OF SECTION 9 ==========================*/
```

Section 9 is a standard windows message handler for our test/example program.
during development we will be adding code to the timer #42 section to update our controls.

CCT_lib.iwb

The CCT_lib file will contain code that actually defines and creates our control as we develop it. Once we are through with development we will convert the file to a static library file.

The following shows the code in the different parts of the skeleton.

Code:
```
/*=======================================================
*       CCT_lib.iwb
*       Custom Control Tutorial Library Source File
*  PART 01
*       Copyright (c)T.L. McCaughn 2011
*  This file may be freely shared
*=======================================================*/
```

This file also has a title block as previously discussed. Notice that this one is also identified as Part 1. The two files will appear as a set as we go through the different stages of development.

Code:
```
        AUTODEFINE "OFF"            /* I always use this - personal preference */
        $include "windowssdk.inc"  /* I always use this - personal preference */
        $include "gdiplus.inc"      /* Needed in order to use the GDI+ library */
```

The AUTODEFINE and first $INCLUDE are the same as previously described. The $INCLUDE "gdiplus.inc" file is required since we will be using the OS's GDI+ library to draw our graphics for our control.

Code:
```
/*---------------SECTION 1-----------------------------
 This section contains the required export declarations for our library
-------------------------------------------------------*/


/*=============== END OF SECTION 1 =====================*/
```

Section 1 will contain our EXPORT function declarations for each of the functions in our control library that a user can invoke. Not all functions within our library will be callable by the user. Some functions will be for internal use only.

Code:
```
/*---------------SECTION 2-----------------------------
 This section contains declarations needed for our library
 along with any needed initialization
-------------------------------------------------------*/

/*=============== END OF SECTION 2 =====================*/
```

Section 2 will contain any definitions and declarations that our control will need.

Code:
```
/*----------------SECTION 3-----------------------------
 This section contains the control's message handler
------------------------------------------------------------*/
SUB MyProc_CC(hWnd:UINT, uMsg:UINT, wParam:UINT, lParam:ANYTYPE),UINT

        SELECT uMsg
                CASE WM_CREATE

                        RETURN 0
      CASE WM_DESTROY

                        RETURN 0
      CASE WM_PAINT

                        RETURN 0
      CASE WM_ERASEBKGND

                        RETURN 0
      CASE WM_SETFOCUS

                        RETURN 0
      CASE WM_KILLFOCUS

                        RETURN 0
        ENDSELECT
   RETURN(DefWindowProc(hWnd, uMsg, wParam, lParam))
ENDSUB
/*=============== END OF SECTION 3 ====================*/
```

Section 3 is the heart of our custom control; the message handler. Notice that the passed parameters match our previous discussions about messages. The OS messages that we will be writing code for are already identified. We will be adding code to all of those. In addition we will be adding some messages of our own creations along with their code.
Notice the RETURN just before the ENDSUB statement. We had stated previously that messages we chose not to handle still had to be handle. This return command passes any unhandled message on the the OS's default window handler.

Obviously, since this is the heart of the control, this handler routine will be rather large before we are through.

Code:
```
/*----------------SECTION 4-----------------------------
 This section contains the routine used to register the control with the OS.
------------------------------------------------------------*/

/*=============== END OF SECTION 4 ====================*/
```

Section 4 is where we will place the code for registering our control with the operating system.

Code:
```
/*----------------SECTION 5-----------------------------
 This section contains the command used to create a control.
------------------------------------------------------------*/

/*=============== END OF SECTION 5 ====================*/
```

Section 5 will contain the code we will use to create an instance of our control.

Code:
```
/*----------------SECTION 6-----------------------------
 This section contains the control's configuration commands
------------------------------------------------------------*/

/*=============== END OF SECTION 6 ================*/
```

Section 6 will contain the code we will use to configure our control.

Code:
```
/*----------------SECTION 7-----------------------------
 This section contains routines for updating the pointer positions of the control
------------------------------------------------------------*/

/*=============== END OF SECTION 7 ====================*/
```

Section 7 will contain the functions that are called by the timer routine in our test program. These functions will cause our control pointer to be updated.

Code:
```
/*----------------SECTION 8-----------------------------
 This section contains routines that support the control's message handler
------------------------------------------------------------*/

/*=============== END OF SECTION 8 ====================*/
```

Section 8 will contain routines that support the message handler directly.

Code:
```
/*---------------SECTION 9-------------------------------
 This section contains routine for the control to talk to its parent
-------------------------------------------------------------*/

/*=============== END OF SECTION 9 ======================*/
```

Section 9 will contain the code that allows our control to send notifications to its parent window.

Code:
```
/*---------------SECTION 10------------------------------
 This section contains any required utility routines
-------------------------------------------------------------/

/*=============== END OF SECTION 10 ======================*/
```

Section 10 will contain generic utility routines required by our control.

That concludes the overview of our skeleton files. A copy of this could be set aside and used as the basis for any custom control the reader may want to attempt.

_____

Coming Next - TBD

IonicWind Software

IWBasic => Create a Custom Control => Topic started by: LarryMc on August 18, 2011, 11:11:10 AM

Title: 12. Control Commands
Post by: LarryMc on August 18, 2011, 11:11:10 AM

There are several places we can start in the construction of our custom control. This is partially due to the well defined goal of what our control should do and what it should look like. If you go on to develop a custom control of your own you may find that a different starting point (from what I will be using in this tutorial) works better for you.

At this point, I feel that the sequence I have started with will be easier for me to explain with some degree of continuity from step to step. The following lists the functional commands to be used with our control:

- Creation Command (including registering our class)

_____

Comming Next - Creation Command

Note:As additional sub-sections are added the list above will be edited to reflect each addition.

IonicWind Software

IWBasic => Create a Custom Control => Topic started by: LarryMc on August 18, 2011, 08:55:50 PM

Title: 13. Creation Command
Post by: LarryMc on August 18, 2011, 08:55:50 PM

This section will cover the command that the user will use to invoke an instance of our control.
We can't use the IWBasic CONTROL command because that command only works with controls that are support internally by IWBasic.

IWBasic has the CONTROLEX command which was designed to handle controls that aren't covered by the CONTROL command.

We also have the option to go with the CreateWindow API command.

I usually try to stay with IWBasic commands when I can. I find they usually save me from extra coding and it a lot of cases have additional safeguards built in.

So, for our purposes, using CONTROLEX is the way to go. Our typical command will look like this:

Code:
```
CONTROLEX win, "LM_Gage1_Class", "", l, t, d, d, 0, 0, id
```

From our previous review we know the following:
win - the control's parent window variable
"LM_Gage1_Class"- the unique name for the class our control is associated with
"" - a blank title string
l - the left edge of our control
t - The top edge of our control
d - The width of our control
d - The height of our control
0 - style_flags
0 - exStyle_flags
id - control identifier unique to its parent.

What's the deal with the two d's, you ask. Since our control is going to be a round gage that means its enclosing box is a rectangle with its width and height equal to the diameter of the gage. And, as for the flags both being 0. That can be due to either not having gotten to them yet or we're not going to use them. It is the latter. We're not going to use them.

Pretty simple... But wait. Back in our review we talked about the need to 'register' a *class* before we can create a window/control that belongs to that *class*.
So we need another function to register our *class* before we create an instance of our control. Our code now looks like this:

Code:
```
        RegisterGage1ClassLM()
        CONTROLEX win,"LM_Gage1_Class","",l,t,d,d,0,0,id
```

We picked a unique name for our new function. We'll cover what goes in it in a few minutes.

So with this scheme the user has to enter two lines of code to create a single control. IWBasic only needs one line for the user to create a control. So, let's do what IWBasic does; put these two lines inside a subroutine. Modifying the above we get:

Code:
```
SUB CreateGageRLM(win as WINDOW,l as INT,t as INT,d as INT,id as UINT )
        RegisterGage1ClassLM()
        CONTROLEX win,"LM_Gage1_Class","",l,t,d,d,0,0,id
   RETURN
ENDSUB
```

The subroutine name has to unique enough to not potentially conflict with any other routine name out in the world. If you duplicate an existing name your potential users might be using you will force them to modify the include file we are going to furnish them in order to 'alias' our routine names. So you need to strike a balance between the name being descriptive of its function, uniqueness, and length.

Notice that we have reduced the number of parameters being passed. There is no need to pass something twice or something we aren't going to use. We only need to pass the parent window, the upper left corner coordinates, the diameter of the gage, and the control's ID.

We'll put our finished subroutine in Section 5 of the CCT_lib.iwb file. It is the only thing that goes in Section 5. But we are still not through with this subroutine.

We have to add the code that will allow the user to access the function.

When a static library file is created, all the functions that are to be available to the user have to be declared as such. It is done with the GLOBAL declaration. Therefore, we need to add the following line of code:

Code:
```
GLOBAL CreateGageRLM
```

We will put this line in Section 1 of the CCT_lib.iwb file. We will put more lines there later.

For the user to call the function from their application they will need a standard subroutine declaration. That line of code looks like this:

Code:
```
DECLARE EXTERN CreateGageRLM(win as WINDOW,l as INT,t as INT,d as INT,id as UINT )
```

This line is placed in Section 1 of the CCT_test.iwb file. We will put more lines there later also. This section will be turned into an include file when development is completed and furnished to the user with the library file.

Now we turn our attention to our registering subroutine we called above.

Code:
```
SUB RegisterGage1ClassLM()

ENDSUB
```

We'll start our discussion by saying that regardless of how many instances of our control are created in a user's application we only need to register our *class* one

time.

From the way we constructed our CreateGageRLM function above the RegisterGage1ClassLM function will be called every time the user creates a new instance of our control. So the first thing we need to do is fix the the internals of the register function so our *class* is only registered prior to the creation of the first instance of our control. It is really simple, with the recent release of IWBasic 2.x. We modify our function as shown:

Code:
```
SUB RegisterGage1ClassLM()
   STATIC INT classRegistered = 0
   IF classRegistered = 0

      classRegistered = 1
   ENDIF
ENDSUB
```

We've created a STATIC variable and initialized it to 0. The name is of no importance as long as it is unique to the subroutine and is not defined elsewhere as a GLOBAL variable. For those who haven't used a static variable at the time of reading this, it functionally works this way.

When the subroutine is called the very first time the variable is set to 0 when the STATIC line of code is encountered. When the subroutine is exited the value of the variable is remembered. On subsequent calls of the subroutine the STATIC line of code is ignored. So the value of the variable is whatever it was the last time the sub was called. With the code structure shown, the first time our routine is called the IF statement will be TRUE. Our registration will be inside the IF/ENDIF block so it will be executed. The last statement sets our variable to 1 (or any non-zero value). When subsequent calls are made the IF statement will be FALSE since the STATIC variable will not be equal to 0.

Having that resolved we now turn our attention to the actual registration.

The OS has a dedicated API function for registering a class; RegisterClassEx. A single system UDT type variable is passed to the function. The following is the definition of the UDT structure:

Code:
```
TYPE WNDCLASSEX
        DEF cbSize AS INT
        DEF style AS INT
        DEF lpfnWndProc AS INT
        DEF cbClsExtra AS INT
        DEF cbWndExtra AS INT
        DEF hInstance AS INT
        DEF hIcon AS INT
        DEF hCursor AS INT
        DEF hbrBackground AS INT
        DEF lpszMenuName AS POINTER
        DEF lpszClassName AS POINTER
        DEF hIconSm AS INT
ENDTYPE
```

Since we are using the "windowssdk.inc" include file we don't have to worry about putting this code in our project. If we created a variable of type WNDCLASSEX and assign the proper values to the necessary elements we can pass the variable to the API function. If it is successful the function will return a non-zero value. Our resulting function will look like this:

Code:
```
SUB RegisterGage1ClassLM()
```

```
   STATIC INT classRegistered = 0
   IF classRegistered = 0
      WNDCLASSEX wc
      wc.cbSize         = len(WNDCLASSEX)
      wc.style          = CS_GLOBALCLASS | CS_HREDRAW | CS_VREDRAW
      wc.lpfnWndProc    = &MyProc_CC
      wc.cbClsExtra     = 0
      wc.cbWndExtra     = 0
      wc.hInstance      = _hinstance
      wc.hIcon          = NULL
      wc.hCursor        = LoadCursor(NULL, IDC_ARROW)
      wc.hbrBackground  = GetStockObject(WHITE_BRUSH)
      wc.lpszMenuName   = NULL
      wc.lpszClassName  = "LM_Gage1_Class"
      wc.hIconSm        = NULL
      classRegistered = RegisterClassEx(wc)
   ENDIF
ENDSUB
```

The following lists each element of the WNDCLASSEX structure along with the value we have assigned that element:

cbSize - len(WNDCLASSEX)
The size, in bytes, of the WNDCLASSEX structure.

style - CS_GLOBALCLASS | CS_HREDRAW |CS_VREDRAW
The class style flags that define how to update the control after moving or resizing it, how to process double-clicks of the mouse, how to allocate space for the device context, and other aspects of the window.

lpfnWndProc - &MyProc_CC
Pointer to the function that processes all messages sent to controls in the class and defines the behavior of the control. This is the name of our handler routine in Section 3 of the CCT_lib.iwb file.

cbClsExtra - 0
Defines amount of extra memory to dedicate to the class. We're not using it.

cbWndExtra - 0
Defines amount of extra memory to dedicate to the control. We're not using it.

hInstance - _hinstance
Identifies the handle to the process that is registering our class. *_hinstance* is an IWBasic global variable that contains the handle.

hIcon - NULL
We are not associating a large icon with our control.

hCursor - LoadCursor(NULL, IDC_ARROW)
Defines the cursor that will be shown when it is over our control.

hbrBackground - GetStockObject(WHITE_BRUSH)
Used to prepare the background of our control if we do not handle it in our handler.

lpszMenuName - NULL
Our control has no menu associated with it.

lpszClassName - "LM_Gage1_Class"
A process unique name for our *class*.

hIconSm - NULL
We are not associating a small icon with our control.

If our call to RegisterClassEx is successful it returns a non-zero value. Otherwise it will return 0.
Because of this we can change our original line of code:
Code:

```
classRegistered = 1
```

to
Code:

```
classRegistered = RegisterClassEx(wc)
```

All that remains is to place our completed registration routine in Section 4 of the CCT_lib.iwb file. It is the only thing that goes in Section 4. Nothing else is required in support of the registration function since it is for internal use only.

_____

Coming Next - Control Configuration

IonicWind Software

IWBasic => Create a Custom Control => Topic started by: LarryMc on August 24, 2011, 09:58:16 PM

Title: 14. Control Configuration
Post by: LarryMc on August 24, 2011, 09:58:16 PM

In the previous section, we got our control's class registered with the OS and we made a way to create an instance of our control.

Now we need to configure our control. This is what makes one instance of our control distinct from all other instances of our control.

In the User's Control Spec section we had the following list:

Next we need to identify the things we need to be able to configure/control:
set the style/type of gage
set the overall size of the gage
set the max min values for the raw data sent to a gage
set the max min values for the displayed range on a gage
set the text identifier for a gage/pointers
set the range multiplier/units text for a gage/pointers, if appropriate
set the offset of clocks to show time zones other than local time.
set the color of rectangle containing the gage
set the white/grey color of the dial face
set the current position of pointer(s), where appropriate

Note: In the above version of the list we have added numbers to facilitate the following discussion.

Item #2 was taken care of in the CreateGageRLM with our left, top, and diameter parameters that were passed to it.

What we want now is a command that can address as many of the remaining list items as we can and that are also practical to set at this point.

We'll start by giving our configuration function a name. As is the case with all our library functions that can be called by the user the name needs to be really unique.

Code:
```
SUB ConfigGageRLM( )

RETURN
```

The following lists each parameter we will pass, along with the item number it addresses in the above list, if appropriate.
win as WINDOW
Required to identify the parent window of this instance of the control.
style as INT
list item #1
title as STRING
list item #5
u as STRING
list item #6
rawmin as INT
list item #3
rawmax as INT
list item #3
dialmin as INT
list item #4
dialmax as INT
list item #4 (For clocks this is used for list item #7)
sf as INT
list item #6
id as UINT
Required to identify this instance of the control.

Inserting the parameters gives us:
Code:
```
SUB ConfigGageRLM(win as WINDOW,_
                  style as INT,_
                  title as STRING,_
                  u as STRING,_
                  rawmin as INT,_
                  rawmax as INT,_
                  dialmin as INT,_
                  dialmax as INT,_
                  sf as INT,_
                  id as UINT )

   RETURN
ENDSUB
```

Two things should be very clear at this point of the tutorial. One, that our control's message handler IS our control and two, that the only way we can communicate with the control is via messages.

There are several ways to get the above parameter values to our control. And, at least one way can be done without sending any messages. However, I believe that the sending of messages is the simplest to understand considering all our discussions to this point.

They can be addressed in any order but we will implement them in the order they are listed. We will use the standard IWBasic SENDMESSAGE command (that you're growing tired of) to send our values. The resulting code is added to our configuration function:
Code:
```
SUB ConfigGageRLM(win as WINDOW,_
                  style as INT,_
                  title as STRING,_
                  u as STRING,_
                  rawmin as INT,_
                  rawmax as INT,_
                  dialmin as INT,_
                  dialmax as INT,_
                  sf as INT,_
                  id as UINT )
```

```
   SENDMESSAGE(win, GAGE_STYLE, 0,style, id)
   SENDMESSAGE(win, GAGE_TITLE, 0,title, id)
   SENDMESSAGE(win, GAGE_UNITS, 0,u, id)
   SENDMESSAGE(win, GAGE_RAWRNG, rawmin,rawmax, id)
   SENDMESSAGE(win, GAGE_DIALRNG, dialmin,dialmax, id)
   SENDMESSAGE(win, GAGE_MULTI, 0,sf, id)
   RETURN
ENDSUB
```

The logical question is where did the GAGE_ message ID constants come from. Right now out of thin air. We need to define them. We do that with the following code:

Code:
```
ENUM GMsg
        GAGE_STYLE      = 0x401
        GAGE_TITLE
        GAGE_UNITS
        GAGE_RAWRNG
        GAGE_DIALRNG
        GAGE_MULTI
        GAGE_SETPANEL_COLOR
        GAGE_DIAL_DARK
        GAGE_CLKOFFSET
        GAGE_SETPOS1
        GAGE_SETPOS2
ENDENUM
```

For those who haven't used ENUM before it is merely an easy way to assign a series of variable constants.
Each ENUM block has to have a name; in this case we picked 'GMsg'. We assign a value to the first item in the list and ENUM assigns the rest is sequential order after that. The fact that we chose to start our message ID constants at 0x401 is not a random act.

When we assign message ID values we have to be absolutely sure that we are not giving our messages the same values as ones the OS is using like WM_CREATE or WM_CLOSE. The OS has set aside the numbers at 0x400 and up for user defined messages.

The above ENUM block is added to SECTION 2 of the CCT_lib.iwb file. Since the user will be unaware that these even exist there is no need to add them to the CCT_test.iwb file.

We went ahead and defined the bottom five constants in the ENUM block since we will address them before this section is complete.

While we're at this point addressing the message ID constants we might as well take care of one other thing.
When we create messages specific for our control we have every expectation that we will use them. In order to use them they have to be added to our message handle routine. The following shows all of our messages added to the handler skeleton in SECTION 3 of the CCT_lib.iwb file:

Code:
```
SUB MyProc_CC(hWnd:UINT, uMsg:UINT, wParam:UINT, lParam:ANYTYPE),UINT

        SELECT uMsg
                CASE WM_CREATE

                        RETURN 0
                CASE WM_DESTROY

                        RETURN 0
                CASE WM_PAINT

                        RETURN 0
                CASE WM_ERASEBKGND

                        RETURN 0
                CASE WM_SETFOCUS

                        RETURN 0
                CASE WM_KILLFOCUS

                        RETURN 0
                CASE GAGE_STYLE

                        RETURN 0
                CASE GAGE_TITLE

                        RETURN 0
                CASE GAGE_UNITS

                        RETURN 0
                CASE GAGE_RAWRNG

                        RETURN 0
                CASE GAGE_DIALRNG

                        RETURN 0
                CASE GAGE_MULTI

                        RETURN 0
                CASE GAGE_SETPANEL_COLOR

                        RETURN 0
                CASE GAGE_DIAL_DARK

                        RETURN 0
                CASE GAGE_CLKOFFSET

                        RETURN 0
                CASE GAGE_SETPOS1

                        RETURN 0
                CASE GAGE_SETPOS2

                        RETURN 0
        ENDSELECT
        RETURN(DefWindowProc(hWnd, uMsg, wParam, lParam))
ENDSUB
```

Okay, back to the ConfigGageRLM command. Since it will be called by the user we need to add the GLOBAL statement to Section 1 of CCT_lib.iwb which results in the section looking like this:

Code:
```
        GLOBAL CreateGageRLM
        GLOBAL ConfigGageRLM
```

We also need to add the declaration statement for it in Section 1 of CCT_test.iwb which results in that section looking like this:

Code:
```
        '$USE "myControl.lib            /* uncomment when converted to include file */
        DECLARE EXTERN CreateGageRLM(win as WINDOW,l as INT,t as INT,d as INT,id as UINT )
        DECLARE EXTERN ConfigGageRLM(win as WINDOW,style as int,title as string,u as string,rawmin as int,rawmax as int,dialmin as int,dialmax as int,sf as int,id as UINT )
```

It appears we are through with the ConfigGageRLM command. Not even close.
This is where the quality of our User's Control Spec comes into play. First we have to account for our different styles of gages. We need to define style constants the user can pass to the ConfigGageRLM command to tell our message handler which type of gage to draw. These style constants are no different than the style_flags we use when we create controls with the CONTROL command.

From the list we made in the User's Control Spec section we arrive at the following:

Code:
```
        CONST ROUND270B = 1
        CONST ROUND270T = 2
        CONST ROUND270L = 3
        CONST ROUND270R = 4
        CONST ROUND360SGL       = 5
        CONST ROUND360DBL       = 6
        CONST ROUNDCLOCK12      = 7
        CONST ROUNDCLOCK24      = 8
        CONST ROUND2PENV        = 9
        CONST ROUND2PENH        = 10
```

Notice that instead of using the ENUM block, as before, we chose to use the CONST declaration. We could have used ENUM. Remember that we have to give the block a name. Before when we used it, the declaration was only in the CCT_lib.iwb file. So there was no possibly of a naming conflict. But these constants will be used directly by the user. So, to prevent any possibility of name conflicts, we didn't use ENUM. It's really nothing significant either way.

We'll add the above code to Section 2 of the CCT_lib.iwb file

Code:
```
        ENUM GMsg
                GAGE_STYLE      = 0x401
                GAGE_TITLE
                GAGE_UNITS
                GAGE_RAWRNG
                GAGE_DIALRNG
                GAGE_MULTI
                GAGE_SETPANEL_COLOR
                GAGE_DIAL_DARK
                GAGE_CLKOFFSET
                GAGE_SETPOS1
                GAGE_SETPOS2
        ENDENUM
        const ROUND270B = 1
        const ROUND270T = 2
        const ROUND270L = 3
        const ROUND270R = 4
        const ROUND360SGL       = 5
        const ROUND360DBL       = 6
        const ROUNDCLOCK12      = 7
        const ROUNDCLOCK24      = 8
        const ROUND2PENV        = 9
        const ROUND2PENH        = 10
```

and Section 1 of the CCT_test.iwb file.

Code:
```
        '$USE "myControl.lib    /* uncomment when converted to include file */
        DECLARE EXTERN CreateGageRLM(win as WINDOW,l as INT,t as INT,d as INT,id as UINT )
        DECLARE EXTERN ConfigGageRLM(win as WINDOW,style as int,title as string,u as string,rawmin as int,rawmax as int,dialmin as int,dialmax as int,sf as int,id as UINT )

        const ROUND270B = 1
        const ROUND270T = 2
        const ROUND270L = 3
        const ROUND270R = 4
        const ROUND360SGL       = 5
        const ROUND360DBL       = 6
        const ROUNDCLOCK12      = 7
        const ROUNDCLOCK24      = 8
        const ROUND2PENV        = 9
        const ROUND2PENH        = 10
```

Now, we will take care of some things in our ConfigGageRLM function that normally would be discovered during the course of developing the code in our message handler routine. But, since we already know what they are (since we are using an existing custom control) we'll cover them all in this section.

The way our code exists right now it is assuming that all our different types of gages use the same parameters. Also, there are no restrictions of any kind on what the user can pass as parameter values other than the type of value.

We'll go about this discussion by addressing each of the SENDMESSAGE commands we have already coded in the order we coded them(shown below):

Code:
```
   SENDMESSAGE(win, GAGE_STYLE, 0,style, id)
   SENDMESSAGE(win, GAGE_TITLE, 0,title, id)
   SENDMESSAGE(win, GAGE_UNITS, 0,u, id)
   SENDMESSAGE(win, GAGE_RAWRNG, rawmin,rawmax, id)
   SENDMESSAGE(win, GAGE_DIALRNG, dialmin,dialmax, id)
   SENDMESSAGE(win, GAGE_MULTI, 0,sf, id)
```

GAGE_STYLE

There is nothing we need to do to it although we could add code to test the value of style to see if it indeed one of the values we have defined.

GAGE_TITLE

Although there is nothing specifically needed to be done to the title we are going to use it to inform us of another problem.

The four 270 degree gages have a restriction placed on them that minimum dial range can be one unit (0 to 1; -15 to -14; 43 to 44; etc) and that the maximum dial range can be fifteen units (0 to 15; -15 to 0; -3 to 12; etc.). If the user passes values that do not meet these criteria we're going to change the title of the gage to read "Dial Min/Max Error!" and we'll set the value to the default(0 to 1).

The following is what the resulting code will look like:

Code:
```
   SENDMESSAGE(win, GAGE_STYLE, 0,style, id)
   STRING ttemp=title
   select style
      case ROUND270B
      case& ROUND270T
      case& ROUND270R
      case& ROUND270L
         if (dialmax-dialmin < 1) | (dialmax-dialmin > 15)
            dialmin=0 : dialmax=1
            ttemp="Dial Min/Max Error!"
```

```
      endif
   endselect
   SENDMESSAGE(win, GAGE_TITLE, 0,ttemp, id)
   SENDMESSAGE(win, GAGE_UNITS, 0,u, id)
   SENDMESSAGE(win, GAGE_RAWRNG, rawmin,rawmax, id)
   SENDMESSAGE(win, GAGE_DIALRNG, dialmin,dialmax, id)
   SENDMESSAGE(win, GAGE_MULTI, 0,sf, id)
```

GAGE_UNITS

No changes or conditions required.

GAGE_RAWRNG

All of our gage types need a set of values for the range of the raw data except for the two clock types. Also, clocks do not need a set of dial range values because their dials are preset. What our clocks do need is an offset value so that we can display different time zones. And, instead of creating a new parameter just for the offset we'll use an existing one that won't be used otherwise for clocks. We'll let the user pass the offset in as the dialmin parameter. And we will send that value with one of our 5 remaining predefined message IDs; GAGE_CLKOFFSET(which takes care of item #7 in our list at the start of this section).
We can handle all of this in a single SELECT block. The resulting code now looks like this:

Code:

```
   SENDMESSAGE(win, GAGE_STYLE, 0,style, id)
   STRING ttemp=title
   select style
      case ROUND270B
      case& ROUND270T
      case& ROUND270R
      case& ROUND270L
         if (dialmax-dialmin < 1) | (dialmax-dialmin > 15)
            dialmin=0 : dialmax=1
            ttemp="Dial Min/Max Error!"
         endif
   endselect
   SENDMESSAGE(win, GAGE_TITLE, 0,ttemp, id)
   SENDMESSAGE(win, GAGE_UNITS, 0,u, id)
   SELECT style
      CASE ROUNDCLOCK12
      CASE& ROUNDCLOCK24
         SENDMESSAGE(win, GAGE_CLKOFFSET, dialmin,0, id)
      DEFAULT
         SENDMESSAGE(win, GAGE_RAWRNG, rawmin,rawmax, id)
   ENDSELECT
   SENDMESSAGE(win, GAGE_DIALRNG, dialmin,dialmax, id)
   SENDMESSAGE(win, GAGE_MULTI, 0,sf, id)
```

GAGE_DIALRNG

We've already discussed above that the four 270 degree gages need a set of min/max dial range values from the user. Also, that the two clocks were fixed. All the rest will be predefined values which the user can't change. This was done to eliminate a lot of display spacing and text size issues. After adding the code to support this we have:

Code:

```
   SENDMESSAGE(win, GAGE_STYLE, 0,style, id)
   STRING ttemp=title
   select style
      case ROUND270B
      case& ROUND270T
      case& ROUND270R
      case& ROUND270L
         if (dialmax-dialmin < 1) | (dialmax-dialmin > 15)
            dialmin=0 : dialmax=1
            ttemp="Dial Min/Max Error!"
         endif
   endselect
   SENDMESSAGE(win, GAGE_TITLE, 0,ttemp, id)
   SENDMESSAGE(win, GAGE_UNITS, 0,u, id)
   SELECT style
      CASE ROUNDCLOCK12
      CASE& ROUNDCLOCK24
         SENDMESSAGE(win, GAGE_CLKOFFSET, dialmin,0, id)
      DEFAULT
         SENDMESSAGE(win, GAGE_RAWRNG, rawmin,rawmax, id)
   ENDSELECT
   SELECT style
      CASE ROUND360SGL
      CASE& ROUND360DBL
      CASE& ROUND2PENV
      CASE& ROUND2PENH
         dialmin = 0 : dialmax = 10
   ENDSELECT
   SENDMESSAGE(win, GAGE_DIALRNG, dialmin,dialmax, id)
   SENDMESSAGE(win, GAGE_MULTI, 0,sf, id)
```

GAGE_MULTI

No changes or conditions required.

Before leaving the ConfigGageRLM command we need to add one more line of code. This line calls a function we will address later:

Code:

```
GageInvalidateBackground(win, id)
```

so our final ConfigGageRLM looks likes this:

Code:

```
SUB ConfigGageRLM(win as WINDOW,_
                  style as INT,_
                  title as STRING,_
                  u as STRING,_
                  rawmin as INT,_
                  rawmax as INT,_
                  dialmin as INT,_
                  dialmax as INT,_
                  sf as INT,_
                  id as UINT )
   SENDMESSAGE(win, GAGE_STYLE, 0,style, id)
   STRING ttemp=title
   select style
      case ROUND270B
      case& ROUND270T
      case& ROUND270R
```

```
      case& ROUND270L
         if (dialmax-dialmin < 1) | (dialmax-dialmin > 15)
            dialmin=0 : dialmax=1
            ttemp="Dial Min/Max Error!"
         endif
   endselect
   SENDMESSAGE(win, GAGE_TITLE, 0,ttemp, id)
   SENDMESSAGE(win, GAGE_UNITS, 0,u, id)
   SELECT style
      CASE ROUNDCLOCK12
      CASE& ROUNDCLOCK24
         SENDMESSAGE(win, GAGE_CLKOFFSET, dialmin,0, id)
      DEFAULT
         SENDMESSAGE(win, GAGE_RAWRNG, rawmin,rawmax, id)
   ENDSELECT
   SELECT style
      CASE ROUND360SGL
      CASE& ROUND360DBL
      CASE& ROUND2PENV
      CASE& ROUND2PENH
         dialmin = 0 : dialmax = 10
   ENDSELECT
   SENDMESSAGE(win, GAGE_DIALRNG, dialmin,dialmax, id)
   SENDMESSAGE(win, GAGE_MULTI, 0,sf, id)
   GageInvalidateBackground(win, id)
   RETURN
ENDSUB
```

We'll place our completed function in Section 6 of the CCT_lib.iwb file.
While we at it we'll create our shell for the GageInvalidateBackground routine:

Code:

```
SUB GageInvalidateBackground(win as WINDOW, id as UINT)

ENDSUB
```

and place it in Section 8 of the CCT_lib.iwb file. We'll cover it in detail later while we're fleshing out the code. For now we'll just say that it is used to insure that our control gets updated properly when the configuration is changed.

We'll finish up this section with our four remaining messages. Each one of them is wrapped in a subroutine callable by the user.

Code:

```
SUB SetGagePanelColorRLM(win as WINDOW,id as UINT,pnlcolor as UINT)
        SENDMESSAGE(win, GAGE_SETPANEL_COLOR, pnlcolor,0, id)
        RETURN
ENDSUB
```

When we create a control it is in a rectangle. With the SetGagePanelColorRLM function the user will be able to set the color of that rectangle to any desired color. It may match the window background or it may be a contrasting color.

Code:

```
SUB SetGageDialDarkRLM(win as WINDOW,id as UINT)
        SENDMESSAGE(win, GAGE_DIAL_DARK, 0,0, id)
        RETURN
ENDSUB
```

With the SetGageDialDarkRLM function the user will be able to select a grey gage dial face as opposed to the default white dial face.

Code:

```
SUB SetGagePos1RLM(win as WINDOW,id as UINT,pos as INT)
        SENDMESSAGE(win, GAGE_SETPOS1, pos,0, id)
        RETURN
ENDSUB
```

The SetGagePos1RLM function is used to set the primary pen position of all gages except clocks.
It is always the longest pen when there are two pens using the same pivot point. For ROUND2PENV type gages it is the pen on the left. For ROUND2PENH type gages it is the pen on the top.

Code:

```
SUB SetGagePos2RLM(win as WINDOW,id as UINT,pos as INT)
        SENDMESSAGE(win, GAGE_SETPOS2, pos,0, id)
        RETURN
ENDSUB
```

The SetGagePos2RLM function is used to set the secondary pen position of the ROUND2PEN type gages. For ROUND2PENV type gages it is the pen on the right. For ROUND2PENH type gages it is the pen on the bottom.

The preceding four functions take care of items #8, 9 and 10 in our list at the beginning of this section. We can now add these to Section 6 of CCT_lib.iwb. And since the user will be calling these routines they have to be declared as GLOBAL in Section 1 of CCT_lib.iwb. They also have to be declared in Section 1 of CCT_test.iwb.
The current listing of the files after making all the changes are attached.

_____

Coming Next - Saving Passed Parameters

IonicWind Software

IWBasic => Create a Custom Control => Topic started by: LarryMc on August 28, 2011, 11:55:38 AM

Title: 15a. Saving Passed Parameters
Post by: LarryMc on August 28, 2011, 11:55:38 AM

In the previous two sections we generated code to create an instance of our custom control. Then we created functions we could use to configure a specific instance of our control by sending messages. Really pretty straight forward once you understand what is going on.

Now, let's consider this. We can have an application that has hundreds of instances of our custom control. And we previously discussed registering the *class* for our control. And that a *class* defines one, and only one, message handler for all controls of that *class* (type).

Therefore, one message handler routine is responsible for drawing hundreds of instances of our control with each one being different from the next.

At this time two things should be obvious. We have to save the configuration values for each instance of the control and the message handler has to be able to access the proper set of values when they are needed.

The first thing a lot of people would think of is a big array somewhere. But that brings up the issue of dynamically resizing arrays since we have no idea how many instances of our control the user will create. And if we create a big static array then we have created another set of problems; the biggest of which is that we have placed an arbitrary limit on the number of our controls that can be used in a single application. The natural response to that is to make the array extremely large. That's fine but what if the user wants to use only one instance of the control. That's a lot of wasted memory.

Those that are a little more advance programmers might suggest using a linked list to store the values for each control. That would remove any limiting of the number of controls in an application. A potential problem there is timing. Linked list are sequential access. The more controls you use the more time it takes to find the data for a specific instance.

Fortunately the OS has given us two simple methods to resolve this issue. The first method is what IWBasic uses internally when creating IWBasic windows and controls. It centers around what the OS calls window's *properties*. The OS allows the assigning of a property value to a specific instance of a window/control via the OS's SetPropA API function. The following are examples of the two most commonly used:

Code:

```
SetPropA(hwnd,"FGRND",GetSysColor(fg))
SetPropA(hwnd,"BGRND",GetSysColor(bg))
```

Three parameters are passed to the function:
1) the handle to the specific instance of the control;
2) a user defined name to identify the specific property;
3) a UINT type variable that is either the specific value itself (as shown above) or

a pointer to the data.

To retrieve the property later the OS's GetPropA API function is used. Retrieving the above values could look something like this:

Code:
```
                        UINT myfg = GetPropA(hwnd,"FGRND")
                        UINT mybg = GetPropA(hwnd,"BGRND")
```

If we had opted to use this technique to save our configuration values there is one other OS API that we would have to use; RemovePropA.

Code:
```
                        RemovePropA(hwnd,"BGRND")
                        RemovePropA(hwnd,"FGRND")
```

The OS system requires that when you use memory that you give it back when you are through with it. This technique would require that we add the RemovePropA commands in the

Code:
```
CASE @IDDESTROY
```

section of our control's message handler. If you check Section 3 of CCT_lib.iwb you will see that the @IDDESTROY message case is already there. Although we aren't going to use the method just described to store our parameters we will still need to use the @IDDESTROY message.

With the method we are going to use we won't be reading and writing the various parameters on a strictly independent basis. Instead, we are going to store all our parameters in a UDT type variable. So the first thing we need to do is construct our UDT.

As with any IWBasic UDT we have to give it a name. Ours will be GAGEPARAMS. So we start with:

Code:
```
TYPE GAGEPARAMS

ENDTYPE
```

Now we will add comments identifying each of the messages we use to send the parameters. To make our list we will simply look at our message handler and copy the various messages we have already set up to handle. Our UDT now looks like this:

Code:
```
TYPE GAGEPARAMS
        /*GAGE_STYLE*/
        /*GAGE_TITLE*/
        /*GAGE_UNITS*/
        /*GAGE_RAWRNG*/
        /*GAGE_DIALRNG*/
        /*GAGE_MULTI*/
        /*GAGE_SETPANEL_COLOR*/
        /*GAGE_DIAL_DARK*/
        /*GAGE_CLKOFFSET*/
        /*GAGE_SETPOS1*/
        /*GAGE_SETPOS2*/
ENDTYPE
```

Now we will give each element that we need a definition statement.

Code:

```
TYPE GAGEPARAMS
  INT       sStyle              /*GAGE_STYLE*/
  STRING sName              /*GAGE_TITLE*/
  STRING sMultiUnits        /*GAGE_UNITS*/
  INT       nRangeMinAct    /*GAGE_RAWRNG*/
  INT       nRangeMaxAct   /*     "     */
  INT       nRangeMinDial   /*GAGE_DIALRNG*/
  INT       nRangeMaxDial  /*     "      */
  INT       nDialMulti          /*GAGE_MULTI*/
  UINT     nPanelColor       /*GAGE_SETPANEL_COLOR*/
  UINT     nGageColor       /*GAGE_DIAL_DARK*/
  INT       clockoffset        /*GAGE_CLKOFFSET*/
  INT       nPos1               /*GAGE_SETPOS1*/
  INT       nPos2               /*GAGE_SETPOS2*/
ENDTYPE
```

We can go ahead and place this UDT definition in Section 2 of the CCT_lib.iwb file. We don't need to place it in the CCT_test.iwb file since it is for internal use only by our control.

In order to benefit from our UDT we have to define a variable of that type. A proper declaration would be:

Code:

```
GAGEPARAMS g_dat
```

Since the g_dat variable is going to be used in our control's message handler, our normal inclination is to place the declaration where we normally place declarations in subroutines which would look like this:

Code:

```
SUB Gage1Proc_LM(hWnd:UINT, uMsg:UINT, wParam:UINT, lParam:ANYTYPE),UINT
   GAGEPARAMS g_dat
.....
ENDSUB
```

Let's think about what we just did. We created a UDT type variable in a subroutine. We know that each time the subroutine is called the variable will be initialized. So that doesn't appear to be the way to save our passed parameters (since we typically only send them once in the beginning).

Those that are aware of the new features of IWBasic 2.x might suggest we use the STATIC keyword like this:

Code:

```
SUB Gage1Proc_LM(hWnd:UINT, uMsg:UINT, wParam:UINT, lParam:ANYTYPE),UINT
   STATIC GAGEPARAMS g_dat
.....
ENDSUB
```

We know with this code the g_dat variable will be initialized on the first call to the subroutine. Thereafter the variable will contain the values stored during the last call of the sub. Sounding good so far. It preserves our values.

Not so fast. This routine could be used for hundreds of our controls in an application. When we go through and configured all of them, which ones' configuration will be stored the next call that the OS makes to repaint the control. Since there is only one variable it will contain the configuration of the last control we configured. So, in essence, all our controls will look the same as each one is repainted because they will all be drawn using the same data.

What we need is a way to create a variable that contains the configuration for each instance of our control. We need a way to insure that the correct data is used with the proper control when the OS needs to redraw it. And, finally, we need a way to do it without using arrays or linked list like we discussed earlier.

In the Review section we talked about an IWBasic window having a UDT associated with it and that the first element was the OS handle to the created window. It appears that that concept of having a UDT with associated data is not unique to IWBasic. When the OS creates a window (remember that a window and a control are one and the same) it creates some reserved memory locations in which to store data that the OS needs. These locations are read with the GetWindowLong API function and written to with the SetWindowLong API function.

If you have been using IWBasic (or its predecessor) very long you will probably recognize that those commands are used to point to the handler routines when sub-classing a control.

Both commands pass two common parameters: the OS handle to the appropriate window/control and an OS constant that identifies a specific reserved memory location. The memory location we are interested in is identified as GWL_USERDATA. The OS sets aside this memory location for each corresponding window/control to allow the user to tie one piece of information to that specific object. In our case we're going to store a pointer to a variable.

We'll start by changing our g_dat declaration. While we are at it we will also add the required SetWindowLong command. The function excerpt now looks like this:

Code:
```
SUB Gage1Proc_LM(hWnd:UINT, uMsg:UINT, wParam:UINT, lParam:ANYTYPE),UINT
   POINTER     g_dat /*GAGEPARAMS*/
   g_dat = new GAGEPARAMS
   SetWindowLong(hWnd, GWL_USERDATA, g_dat)
.....
ENDSUB
```

We have declared g_dat to be a pointer.
We created a new UDT of type GAGEPARAMS in memory and the pointer to that memory is stored in g_dat.
We have used the SetWindowLong function to store the pointer value in the reserved GWL_USERDATA memory location for the current window.

Problem with the code structured this way is that after running for a while our application would crash. Remember hundreds of controls receiving hundreds of messages. With this setup we're taking a chuck of memory the size of our UDT each time any message is sent to each of our control instances. But this is an easy fix. Our message handler is already set up to handle the WM_CREATE / @IDCREATE message. This message is sent by the OS when a window/control is first created but before it is displayed. So, if we adjust our code structure to the following we will wind up with one UDT pointer stored per control when the control is first created.

Code:
```
SUB Gage1Proc_LM(hWnd:UINT, uMsg:UINT, wParam:UINT, lParam:ANYTYPE),UINT
   POINTER     g_dat /*GAGEPARAMS*/
   SELECT uMsg
      CASE WM_CREATE
         g_dat = NEW GAGEPARAMS
         SetWindowLong(hWnd, GWL_USERDATA, g_dat)
.....
ENDSUB
```

The next step is to obtain a control's specific g_dat UDT each time a message is sent for that instance of the control. We do that with:

Code:
```
g_dat = GetWindowLong(hWnd, GWL_USERDATA)
```

The following shows the proper placement of the above line of code:

Code:
```
SUB Gage1Proc_LM(hWnd:UINT, uMsg:UINT, wParam:UINT, lParam:ANYTYPE),UINT
   POINTER     g_dat /*GAGEPARAMS*/

   g_dat = GetWindowLong(hWnd, GWL_USERDATA)
   SELECT uMsg
      CASE WM_CREATE
         g_dat = NEW GAGEPARAMS
         SetWindowLong(hWnd, GWL_USERDATA, g_dat)
.....
ENDSUB
```

Each time a message is sent hWnd identifies the specific instance of our control. Before proceeding we obtain the pointer to our data. Before WM_CREATE the pointer will be NULL. After WM_CREATE is sent the pointer will always point to the appropriate data.

There is one thing that we ALWAYS have to remember about pointers and the NEW command. When we take memory we have to give it back when we're through with it. If we don't we wind up with what is called 'memory leaks'. Slowly but surely we use up all our memory and when we do, we crash.

When it is time for a control to be deleted the OS sends the WM_DESTROY / @IDDESTROY destroy message. This is done in order to give us the opportunity to do this very kind of cleanup.

The following shows how we return the memory used by g_dat for the current instance of the control.

Code:
```
SUB Gage1Proc_LM(hWnd:UINT, uMsg:UINT, wParam:UINT, lParam:ANYTYPE),UINT
   POINTER     g_dat /*GAGEPARAMS*/

   g_dat = GetWindowLong(hWnd, GWL_USERDATA)
   SELECT uMsg
      CASE WM_CREATE
         g_dat = NEW GAGEPARAMS
         SetWindowLong(hWnd, GWL_USERDATA, g_dat)
         RETURN 0
      CASE WM_DESTROY
         FreeHeap(g_dat)
         RETURN 0
.....
ENDSUB
```

The next step is to establish some default values for the elements in g_dat.

Code:
```
   'init g_dat
   #g_dat.sStyle                = 0
   #g_dat.sName         = ""
   #g_dat.sMultiUnits           = ""
   #g_dat.nRangeMinAct  = 0
   #g_dat.nRangeMaxAct  = 100
   #g_dat.nRangeMinDial         = -1
   #g_dat.nRangeMaxDial         = 14
   #g_dat.nDialMulti            = 10
   int r1,g1,b1
   GageunRGBLM(GetSysColor(0), r1, g1, b1)
```

```
   #g_dat.nPanelColor   = RGBA(r1,g1,b1,255)
   #g_dat.nGageColor            = RGBA(255,255,255,255)
   #g_dat.clockoffset           = 0
```

GageunRGBLM is a utility function that converts a RGB color value into its individual color components. We're doing that here because we need the three values to pass to the RGBA IWBasic function which adds the alpha component (where 0 is fully transparent and 255 is fully opaque). The GDI+ library, which will be used for drawing our control, requires colors be in this format.
We are actually setting the default color of the rectangle that contains our control to the grey color of a scrollbar. And the gage color (dial face) to white.

The following code for the GageunRGBLM function is placed in Section 10 of CCT_lib.iwb:

Code:

```
SUB GageunRGBLM(col as uint, r as int byref, g as int byref, b as int byref)
   r=col%256 :'Red
   g=(col%65536)/256 :'Green
   b=col/65536 :'Blue
   RETURN
ENDSUB
```

The following shows the proper insertion point for our default values (after an instance of g_dat is created):

Code:

```
SUB MyProc_CC(hWnd:UINT, uMsg:UINT, wParam:UINT, lParam:ANYTYPE),UINT
   POINTER     g_dat /*GAGEPARAMS*/

   g_dat = GetWindowLong(hWnd, GWL_USERDATA)
   SELECT uMsg
      CASE WM_CREATE
         g_dat = NEW GAGEPARAMS
         SetWindowLong(hWnd, GWL_USERDATA, g_dat)
         'init g_dat
         #g_dat.sStyle          = 0
         #g_dat.sName                   = ""
         #g_dat.sMultiUnits             = ""
         #g_dat.nRangeMinAct            = 0
         #g_dat.nRangeMaxAct            = 100
         #g_dat.nRangeMinDial   = -1
         #g_dat.nRangeMaxDial   = 14
         #g_dat.nDialMulti              = 10
         int r1,g1,b1
         GageunRGBLM(GetSysColor(0), r1, g1, b1)
         #g_dat.nPanelColor             = RGBA(r1,g1,b1,255)
         #g_dat.nGageColor              = RGBA(255,255,255,255)
         #g_dat.clockoffset             = 0
         RETURN 0
      CASE WM_DESTROY
         FreeHeap(g_dat)
         RETURN 0
......
```

———————————————————

Coming Next - Saving Passed Parameters - Part B

IonicWind Software

IWBasic => Create a Custom Control => Topic started by: LarryMc on August 30, 2011, 10:31:14 PM

Title: 15b. Saving Passed Parameters
Post by: LarryMc on August 30, 2011, 10:31:14 PM

What we have left to do in this section is to add the code for each handler message that receives configuration data and stores it in g_dat.

We'll start with the portion of our handler that contains those messages:

Code:

```
....
      CASE GAGE_STYLE

         RETURN 0
      CASE GAGE_TITLE

         RETURN 0
      CASE GAGE_UNITS

         RETURN 0
      CASE GAGE_RAWRNG

         RETURN 0
      CASE GAGE_DIALRNG

         RETURN 0
      CASE GAGE_MULTI

         RETURN 0
      CASE GAGE_SETPANEL_COLOR

         RETURN 0
      CASE GAGE_DIAL_DARK

         RETURN 0
      CASE GAGE_CLKOFFSET

         RETURN 0
      CASE GAGE_SETPOS1

         RETURN 0
      CASE GAGE_SETPOS2

         RETURN 0
.....
```

For the most part it is simply a matter of loading the wparam or lparam value into the proper g_dat element. But a few are a little more complicated. We'll start with the simplest:

Code:

```
       CASE GAGE_STYLE
         #g_dat.sStyle = #<int>lParam
         RETURN 0
```

Notice that we used #<int>lParam instead of just lParam. If you go back and look at the line where our handler is declared you will see that we defined the lparam parameter as ANYTYPE. That means that each time we read/write that variable we have to specifiy what type of variable it currently contains. If we don't specify or use the wrong type we can start overwriting memory and corrupt or crash the application. We don't have to do this for the wParam parameter because we defined it as a UINT.

Now we will proceed with the simple handler cases:

Code:

```
      CASE GAGE_TITLE
         #g_dat.sName = ##<string>lParam
         RETURN 0
```

Code:

```
      CASE GAGE_UNITS
         #g_dat.sMultiUnits = ##<string>lParam
         RETURN 0
```

Code:

```
      CASE GAGE_RAWRNG
         #g_dat.nRangeMinAct = wParam
         #g_dat.nRangeMaxAct = #<int>lParam
         RETURN 0
```

Code:

```
      CASE GAGE_MULTI
         #g_dat.nDialMulti = #<int>lParam
         RETURN 0
```

Code:

```
     CASE GAGE_CLKOFFSET
         #g_dat.clockoffset = wParam
         RETURN 0
```

All the CASE statements we've covered so far have one thing in common. They were sent from our ConfigGageRLM subroutine. The significance of that is the presence of the GageInvalidateBackground function call at the end of the subroutine. What we said about it at the time was

Quote

```
"... it is used to insure that our control gets updated properly when the configuration is changed.".
```

For the rest of our CASE statements we will have to take care of that ourselves. The following OS function will cause a WM_PAINT message to be sent which will cause the current instance of the control to be redrawn:

Code:

```
         InvalidateRect(hWnd, NULL, FALSE)
```

The rest of our CASE statements will include the above call.

Code:

```
      CASE GAGE_SETPANEL_COLOR
         #g_dat.nPanelColor = wParam
         InvalidateRect(hWnd, NULL, FALSE)
         RETURN 0
```

Code:

```
      CASE GAGE_DIAL_DARK
         #g_dat.nGageColor = RGB(140,140,140)
         InvalidateRect(hWnd, NULL, FALSE)
         RETURN 0
```

Code:

```
      CASE GAGE_SETPOS1
         SELECT #g_dat.sStyle
            CASE ROUNDCLOCK12
            CASE& ROUNDCLOCK24
            DEFAULT
               #g_dat.nPos1 = wParam
         ENDSELECT
         InvalidateRect(hWnd, NULL, FALSE)
         RETURN 0
```

Code:

```
      CASE GAGE_SETPOS2
         #g_dat.nPos2 = wParam
         InvalidateRect(hWnd, NULL, FALSE)
         RETURN 0
```

For the CASE GAGE_SETPOS1 block (above) we needed to insert the added SELECT/ENDSELECT block. Since the clock hands are controlled internally by the PC's time we don't need to save the pointer position value. But we will use it to control how often the clocks are forced to redraw.

We saved the most complicated for last. We'll start with the minimum:

Code:

```
      CASE GAGE_DIALRNG
         #g_dat.nRangeMinDial = wParam
         #g_dat.nRangeMaxDial = #<int>lParam
         RETURN 0
```

The rest of the code we are going to add to this CASE statement could be done in the portion of the handler that paints our gage. However, that would be highly inefficient since we really only need to execute the code we're going to add when the gage is configured. So what code are we talking about?

Our custom control is a ROUND gage. As such a lot of our drawing will be based upon arcs of varying sizes.

The placement of tick marks and the end points of pointers are all calculated based upon the sine and cosine of a specified angle. The arc drawing function we will use assumes 0 degrees is at 3 o'clock and the numbers increase in a clockwise direction. We'll start with our 270 degree gages. The standard placement is for the 90 degree empty segment to be center on the bottom of the gage. The other three versions will be rotated by the proper amount from that standard position. Our first added code will be:

Code:
```
   INT rot
   SELECT #g_dat.sStyle
      CASE ROUND270B
         rot=0
      CASE ROUND270T
         rot=180
      CASE ROUND270R
         rot=270
      CASE ROUND270L
         rot=90
   ENDSELECT
```

We've created a local variable to hold the amount of adjustment that needs to be made for each of the four 270 degree gages.

It must be noted that the remaining code we are going to add in this section was developed via trial and error during the course of developing the gage control originally. What we will do here is describe what we did and why we did it.
The next portion of code is another SELECT block:

Code:
```
      select #g_dat.sStyle
         case ROUND270B
         case& ROUND270R
         case& ROUND270L
         case& ROUND270T
            if #g_dat.nRangeMaxDial-#g_dat.nRangeMinDial <1 | #g_dat.nRangeMaxDial-#g_dat.nRangeMinDial >15
               #g_dat.nAngleStart       = -230+rot
               #g_dat.nAngleEnd = 50+rot
               #g_dat.nRangeMinDial = 0
               #g_dat.nRangeMaxDial = 0
            endif
      ...
```

In this code we are checking to see if the user min and max dial range values are such that the range will be between 1 and 15. This is because we want to limit the number of numbers displayed on the dial of this tpe of gage to prevent overcrowding. If the value is outside the acceptable range we set some default values. Let's examine those defaults:

Code:
```
               #g_dat.nAngleStart       = -230+rot
               #g_dat.nAngleEnd = 50+rot
               #g_dat.nRangeMinDial = 0
               #g_dat.nRangeMaxDial = 0
```

If you examine our GAGEPARAMS UDT you will notice there are no elements named nAngleStart and nAngleEnd. The sine/cosine calculations we will use when drawing our gage will need a starting and an ending angle. So we need to add these two elements to our UDT which will now look like this:

Code:
```
TYPE GAGEPARAMS
  INT           sStyle          /*GAGE_STYLE*/
  STRING        sName           /*GAGE_TITLE*/
  STRING        sMultiUnits     /*GAGE_UNITS*/
  INT           nRangeMinAct    /*GAGE_RAWRNG*/
  INT           nRangeMaxAct    /*     "     */
  INT           nRangeMinDial   /*GAGE_DIALRNG*/
  INT           nRangeMaxDial   /*     "      */
  INT           nDialMulti      /*GAGE_MULTI*/
  UINT          nPanelColor     /*GAGE_SETPANEL_COLOR*/
  UINT          nGageColor      /*GAGE_DIAL_DARK*/
  INT           clockoffset     /*GAGE_CLKOFFSET*/
  INT           nPos1           /*GAGE_SETPOS1*/
  INT           nPos2           /*GAGE_SETPOS2*/
  FLOAT         nAngleStart
  FLOAT         nAngleEnd
ENDTYPE
```

Let's examine the following two lines of code closer:

Code:

```
            #g_dat.nAngleStart       = -230+rot
            #g_dat.nAngleEnd      = 50+rot
```

A start angle of -230 degrees puts the min dial value at the 7:00 o'clock position. It's a negative number because the 0 degree position for our drawing function is based at 3:00 o'clock. The end angle of 50 degrees is the max dial value at the 5:00 o'clock position. We then shift the start/stop by the value of rot to account for our 4 different 270 type gages. Doing the algebra we see that our default 270 gage really has a span of 280 degrees. There is a very specific reason for the change.

We limit the number of major tick marks (and numbers) on these gages to 16. Examples would be 0 to 15, -15 to 0, 40 to 55, etc. The choice is up to the user. It is then our task to evenly space those tick marks on our dial. That means that the number of tick marks required has to evenly divide into our span. And it just so happens that most of the numbers between 1 and 15 divide evenly into 280. Where they don't divide evenly we have to adjust the start/stop to gives us a span that they will be evenly spaced.

We will now expand our code to make the necessary adjustments for 270 degree type gages.

Code:

```
      select #g_dat.sStyle
         case ROUND270B
         case& ROUND270R
         case& ROUND270L
         case& ROUND270T
            if #g_dat.nRangeMaxDial-#g_dat.nRangeMinDial <1 | #g_dat.nRangeMaxDial-#g_dat.nRangeMinDial >15
               #g_dat.nAngleStart       = -230+rot
               #g_dat.nAngleEnd = 50+rot
               #g_dat.nRangeMinDial = 0
               #g_dat.nRangeMaxDial = 0
            endif
            select #g_dat.nRangeMaxDial-#g_dat.nRangeMinDial
               case 1
               case& 2
               case& 4
               case& 5
               case& 7
               case& 8
               case& 10
               case& 14
                  '280
                  #g_dat.nAngleStart    = -230+rot
                  #g_dat.nAngleEnd = 50+rot
               case 3
                  '282
                  #g_dat.nAngleStart    = -231+rot
                  #g_dat.nAngleEnd = 51+rot
               case 6
               case& 9
               case& 12
                  '288
                  #g_dat.nAngleStart    = -234+rot
                  #g_dat.nAngleEnd = 54+rot
               case 11
               case& 13
                 '286
                  #g_dat.nAngleStart    = -233+rot
                  #g_dat.nAngleEnd = 53+rot
               case 15
                  #g_dat.nAngleStart    = -225+rot
                  #g_dat.nAngleEnd = 45+rot
            endselect
....
```

Our last adjustment is for the 360 degree gages and the clocks. This adjustment is made to make the starting point appear at the 12:00 o'clock position. The code appears as:

Code:

```
        case ROUND360SGL
        case& ROUND360DBL
        case& ROUNDCLOCK12
        case& ROUNDCLOCK24
           '360
           #g_dat.nAngleStart   = -90
           #g_dat.nAngleEnd = 270
```

The final code for the handler for the GAGE_DIALRNG message is:

Code:

```
               case GAGE_DIALRNG
                  #g_dat.nRangeMinDial = wParam
                  #g_dat.nRangeMaxDial = #<int>lParam
```

```
                  select #g_dat.sStyle
                     case ROUND270B
                       rot=0
                     case ROUND270T
                         rot=180
                     case ROUND270R
                         rot=270
                     case ROUND270L
                        rot=90
                  endselect
                  select #g_dat.sStyle
                     case ROUND270B
                     case& ROUND270R
                     case& ROUND270L
                     case& ROUND270T
                        if #g_dat.nRangeMaxDial-#g_dat.nRangeMinDial <1 | #g_dat.nRangeMaxDial-#g_dat.nRangeMinDial >15
                          #g_dat.nAngleStart   = -230+rot
                          #g_dat.nAngleEnd = 50+rot
                          #g_dat.nRangeMinDial = 0
                          #g_dat.nRangeMaxDial = 0
                        endif
                        select #g_dat.nRangeMaxDial-#g_dat.nRangeMinDial
                           case 1
                           case& 2
                           case& 4
                           case& 5
                           case& 7
                           case& 8
                           case& 10
                           case& 14
                              '280
                                #g_dat.nAngleStart     = -230+rot
                                #g_dat.nAngleEnd = 50+rot
                           case 3
                                '282
                                #g_dat.nAngleStart     = -231+rot
                                #g_dat.nAngleEnd = 51+rot
                           case 6
                           case& 9
                           case& 12
                                '288
                                #g_dat.nAngleStart     = -234+rot
                                #g_dat.nAngleEnd = 54+rot
                           case 11
                           case& 13
                                '286
                                #g_dat.nAngleStart     = -233+rot
                                #g_dat.nAngleEnd = 53+rot
                           case 15
                                #g_dat.nAngleStart     = -225+rot
                                #g_dat.nAngleEnd = 45+rot
                        endselect
                     case ROUND360SGL
                     case& ROUND360DBL
                     case& ROUNDCLOCK12
                     case& ROUNDCLOCK24
                        '360
                        #g_dat.nAngleStart     = -90
                        #g_dat.nAngleEnd = 270
                  endselect
               return 0
```

This completes the section on saving passed parameters.

_____

Coming Next - Graphics Review

IonicWind Software

IWBasic => Create a Custom Control => Topic started by: LarryMc on September 05, 2011, 09:22:53 PM

Title: 16. Graphics Review
Post by: LarryMc on September 05, 2011, 09:22:53 PM

The core OS component responsible for representing graphical objects and transmitting them to output devices such as monitors and printers is the Graphics Device Interface (GDI). GDI is responsible for tasks such as drawing lines and curves, rendering fonts and handling palettes. It is not directly responsible for drawing windows, menus, etc.. Those tasks are performed by the user subsystem, which resides on top of the GDI. The components which connect the GDI to the various devices that can be drawn to are called device drivers.

The OS informs GDI to prepare the device for drawing operations (such as selecting a line color and width, a brush pattern and color, a font typeface, a clipping region, and so on). These tasks are accomplished by creating and maintaining a device context (DC). A DC is a structure that defines a set of graphic objects and their associated attributes, and the graphic modes that affect output. The graphic objects include a pen for line drawing, a brush for painting and filling, a bitmap for copying or scrolling parts of the screen, a palette for defining the set of available colors, a region for clipping and other operations, and a path for painting and drawing operations (see chart below). The actual context is maintained by GDI. A handle to the DC (HDC) is obtained before output is written and then released after elements have been written.
A DC, like most GDI objects, is opaque - its data cannot be accessed directly, but its handle can be passed to various GDI functions that will operate on it, either to draw an object, to retrieve information about it, or to change the object in some way.
When an application creates a DC, the system automatically stores a set of default objects in it. (There is no default bitmap or path.) The application can change these defaults by creating a new object and selecting it into the DC.

Graphic objects and Associated attributes
Bitmap - Size, in bytes; dimensions, in pixels; color-format; compression scheme; and so on.
Brush - Style, color, pattern, and origin.
Palette - Colors and size (or number of colors).
Font - Typeface name, width, height, weight, character set, and so on.
Path - Shape.
Pen - Style, width, and color.
Region - Location and dimensions.

With the introduction of Windows XP, GDI was deprecated in favor of its successor, the GDI+ subsystem. GDI+ adds anti-aliased 2D graphics, floating point coordinates, gradient shading, more complex path management, intrinsic support for modern graphics-file formats like JPEG and PNG, and support for composition of affine transformations in the 2D view pipeline. GDI+ uses ARGB values to represent color.

_____

Coming Next - WM_PAINT Setup

IonicWind Software

IWBasic => Create a Custom Control => Topic started by: LarryMc on September 09, 2011, 07:26:48 PM

Title: 17. WM_PAINT Setup
Post by: LarryMc on September 09, 2011, 07:26:48 PM

We're finally at the point where we are going to set up the code structure for the actual drawing of our control. From earlier discussions we know that the OS sends the WM_PAINT / @IDPAINT message whenever there is a need to redraw our control. That is, when all or a portion of our control display area has been made invalid. That can be caused by moving another object over our control on the screen or by calling the InvalidateRect API function

This is the code excerpt we will start with:

Code:
```
SUB Gage1Proc_LM(hWnd:UINT, uMsg:UINT, wParam:UINT, lParam:ANYTYPE),UINT
...
   PAINTSTRUCT ps
...
   SELECT uMsg
      CASE WM_PAINT
         BeginPaint(hWnd, ps)
         'draw our control

         EndPaint(hWnd, ps)
         RETURN 0
...
```

The BeginPaint API function prepares the current window/control for painting and fills a PAINTSTRUCT type structure with information about the painting. The function is passed two parameters: the handle (hWnd) to the current instance of our control and a previously defined variable (ps) of type PAINTSTRUCT. The only element of the PAINTSTRUCT UDT that we are interested in is the first; hdc. This is the screen device context used for painting our control.

The last API function that we call in the WM_PAINT portion of the handler is EndPaint. It passes the same two parameters as the BeginPaint function. The main purpose of this function, as far as we are concerned, is to validate our control. In other words, it tells the OS that our control has been updated and no further WM_PAINT messages are needed at this time.

Some readers may be asking themselves why don't we use GetDC / ReleaseDC instead of BeginPaint / EndPaint. It all has to do with the invalidating / validating of the client area of our control. The rule of thumb to remember is inside the WM_PAINT handler always use BeginPaint / EndPaint otherwise use GetDC / ReleaseDC.

The code structure above is enough to allow us to start using GDI functions to draw our control. But there is a potential problem. With this arrangement there is the potential for flickering if our drawing code is very complicated (time consuming).
To eliminate the possible problem we will use a technique called "double-buffering".

This is done by creating a compatible device context in memory along with a compatible bitmap in memory and doing all our drawing to memory instead of directly to the screen. When we are through with all our drawing we'll transfer the complete memory bitmap to the screen device context. In that way we update our control on the screen all at one time, flicker free.
The following is how the existing code would need to be modified

Code:
```
...
   PAINTSTRUCT ps
   UINT        hdc2
...
   SELECT uMsg
      CASE WM_PAINT
         BeginPaint(hWnd, ps)
         'create our memory DC that is compatible with our screen DC (ps.hdc)
         hdc2 = CreateCompatibleDC(ps.hdc)
         'create our memory bitmap that is compatible with our screen DC (ps.hdc) and make it 200 x 200 pixels
```

```
        UINT hbmp = SelectObject(hdc2, CreateCompatibleBitmap(ps.hdc, 200, 200))
        'draw our control

        EndPaint(hWnd, ps)
        RETURN 0
....
```

Code:

```
        hdc2 = CreateCompatibleDC(ps.hdc)
```

This line creates a memory DC that is compatible with the screen DC for our control(ps.hdc) and returns the memory DC's handle (hdc2).

Code:

```
        UINT hbmp = SelectObject(hdc2, CreateCompatibleBitmap(ps.hdc, 200, 200))
```

This line creates a memory 200 x 200 pixel bitmap that is compatible with the screen DC for our control(ps.hdc). It then selects the memory bitmap object into the memory DC (hdc2). Said another way, it assigns the memory bitmap to the memory DC.
When finished, hbmp will contain the handle to the bitmap that is being replaced.

The golden rule of windows programming is that when you create something temporarily that uses memory you must free (give back) the memory when you are finished with it. Failure to do so will cause erratic program behavior and ultimately a program crash. The previous two lines of code consumed memory. We now need to add code to give the memory back. We'll start with the memory bitmap. Remember from the SelectObject function above that it returns the handle to the object being replace. We saved the original, replaced bitmap handle in hbmp. So, by selecting it, the function will return the handle to our compatible bitmap that we created. With that handle, we can delete the object thus freeing up the memory. The following line of code accomplishes this for us:

Code:

```
DeleteObject(SelectObject(hdc2, hbmp))
```

It is important to note that an object that is currently selected into (assigned to) a DC can not be deleted. The object must first be de-selected using the technique just demonstrated.

After all objects have been deleted we can delete the memory DC itself. We accomplish this with the following line of code:

Code:

```
DeleteDC(hdc2)
```

Adding these two lines to our handler excerpt we have:

Code:

```
...
   PAINTSTRUCT ps
   UINT        hdc2
...
   SELECT uMsg
      CASE WM_PAINT
         BeginPaint(hWnd, ps)
         'create our memory DC that is compatible with our screen DC (ps.hdc)
         hdc2 = CreateCompatibleDC(ps.hdc)
         'create our memory bitmap that is compatible with our screen DC (ps.hdc) and make it 200 x 200 pixels
         UINT hbmp = SelectObject(hdc2, CreateCompatibleBitmap(ps.hdc, 200, 200))
         'draw our control

         EndPaint(hWnd, ps)
         DeleteObject(SelectObject(hdc2, hbmp))
         DeleteDC(hdc2)
         RETURN 0
....
```

Now that we have our buffer taken care of we need to add the code that transfers what we draw on the memory map to the screen DC. That is done with the following code:

Code:

```
SetStretchBltMode(ps.hdc, HALFTONE)
SetBrushOrgEx(ps.hdc,0, 0,null)
WINRECT     rc
GetClientRect(hWnd, rc)
int rcsize = rc.right
StretchBlt(ps.hdc, 0, 0, rcsize, rcsize, hdc2, 0, 0, 200, 200, SRCCOPY)
```

When we created our memory bitmap we set it to 200 x 200 pixels. This was done so that we could draw all of our gages at one size. This eliminates any rounding errors that might appear due to scaling all our calculations to the control's actual size, set by the user. With the above block of code we are going to scale the entire control to the size requested by the user.

Code:
```
SetStretchBltMode(ps.hdc, HALFTONE)
```
This line sets the bitmap stretching mode in the screen DC to HALFTONE. HALFTONE mode maps pixels from the source rectangle into blocks of pixels in the destination rectangle. The average color over the destination block of pixels approximates the color of the source pixels. Using this mode requires that we use the following function:

Code:
```
SetBrushOrgEx(ps.hdc,0, 0,null)
```
This function sets the brush origin that GDI assigns to the next brush an application selects into the memory DC. The line above sets the default values.

Code:
```
winrect rc
GetClientRect(hWnd, rc)
int rcsize = rc.right
```
These three lines load the dimensions of the controls client area into a WINRECT UDT and then uses the right element for the rcsize value in the next function.

Code:
```
StretchBlt(ps.hdc, 0, 0, rcsize, rcsize, hdc2, 0, 0, 200, 200, SRCCOPY)
```
This function does the actual copying of our memory DC to the screen DC; scaling up/down based upon the value of rcsize which is the radius of the control requested by the user.

Our handler excerpt now looks like the following:

Code:
```
...
   PAINTSTRUCT ps
   UINT        hdc2
...
   SELECT uMsg
      CASE WM_PAINT
         BeginPaint(hWnd, ps)
         'create our memory DC that is compatible with our screen DC (ps.hdc)
         hdc2 = CreateCompatibleDC(ps.hdc)
         'create our memory bitmap that is compatible with our screen DC (ps.hdc) and make it 200 x 200 pixels
         UINT hbmp = SelectObject(hdc2, CreateCompatibleBitmap(ps.hdc, 200, 200))
         'draw our control

         SetStretchBltMode(ps.hdc, HALFTONE)
         SetBrushOrgEx(ps.hdc,0, 0,null)
         WINRECT     rc
         GetClientRect(hWnd, rc)
         int rcsize = rc.right
         StretchBlt(ps.hdc, 0, 0, rcsize, rcsize, hdc2, 0, 0, 200, 200, SRCCOPY)
         EndPaint(hWnd, ps)
         DeleteObject(SelectObject(hdc2, hbmp))
         DeleteDC(hdc2)
         RETURN 0
....
```

With the resulting code block (shown above) we can now start writing the actual drawing code. However, if we do our code will not be very efficient. Each time the WM_PAINT message is received by the handler we are creating a memory DC and bitmap and then deleting it.

There is no reason why we need to do it that way. It would be much more efficient to create the memory DC and bitmap the first time WM_PAINT message is sent to the current instance; save their handles so we can reuse them for all future updates and then give the memory back when we no longer the control.

We're going to modify our code above to make the improvement. We'll start by creating two variables to store the handles of the memory DC and the memory bitmap. Since these variables have to be unique to each instance of our control we will make them elements in our g_dat UDT. We'll start by adding the two new elements to the top of our GAGEPARAMS UDT structure definition in Section 2 of CCT_lib.iwb. The definition will now look like:

Code:

```
TYPE GAGEPARAMS
   UINT dcHand
   UINT bitHand
   INT          sStyle                /*GAGE_STYLE*/
   STRING       sName                 /*GAGE_TITLE*/
   STRING       sMultiUnits           /*GAGE_UNITS*/
   INT          nRangeMinAct          /*GAGE_RAWRNG*/
   INT          nRangeMaxAct          /*     "     */
   INT          nRangeMinDial         /*GAGE_DIALRNG*/
   INT          nRangeMaxDial         /*     "      */
   INT          nDialMulti            /*GAGE_MULTI*/
   UINT         nPanelColor           /*GAGE_SETPANEL_COLOR*/
   UINT         nGageColor            /*GAGE_DIAL_DARK*/
   INT          clockoffset           /*GAGE_CLKOFFSET*/
   INT          nPos1                 /*GAGE_SETPOS1*/
   INT          nPos2                 /*GAGE_SETPOS2*/
   FLOAT        nAngleStart
   FLOAT        nAngleEnd
ENDTYPE
```

Next we'll initialize the elements at the top portion of the WM_CREATE section of our message handler; which will now appear as:

Code:

```
               CASE WM_CREATE
                       g_dat = NEW GAGEPARAMS
                       SetWindowLong(hWnd, GWL_USERDATA, g_dat)
                       'init g_dat
                       #g_dat.dcHand                  = 0
                       #g_dat.bitHand                 = 0
                       #g_dat.sStyle                  = 0
                       #g_dat.sName                   = ""
                       #g_dat.sMultiUnits             = ""
                       #g_dat.nRangeMinAct            = 0
                       #g_dat.nRangeMaxAct            = 100
                       #g_dat.nRangeMinDial           = -1
                       #g_dat.nRangeMaxDial           = 14
                       #g_dat.nDialMulti              = 10
                       int r1,g1,b1
                       GageunRGBLM(GetSysColor(0), r1, g1, b1)
                       #g_dat.nPanelColor             = RGBA(r1,g1,b1,255)
                       #g_dat.nGageColor              = RGBA(255,255,255,255)
                       #g_dat.clockoffset             = 0
                       #g_dat.nAngleStart             = -230
                       #g_dat.nAngleEnd               = 50
                       RETURN 0
```

Next we will modify the following portion of the WM_PAINT code:

Code:

```
      CASE WM_PAINT
         BeginPaint(hWnd, ps)
         'create our memory DC that is compatible with our screen DC (ps.hdc)
         hdc2 = CreateCompatibleDC(ps.hdc)
         'create our memory bitmap that is compatible with our screen DC (ps.hdc) and make it 200 x 200 pixels
         UINT hbmp = SelectObject(hdc2, CreateCompatibleBitmap(ps.hdc, 200, 200))
         'draw our control
```

to look like this:

Code:

```
      CASE WM_PAINT
         BeginPaint(hWnd, ps)
          if #g_dat.dcHand = 0
            'create our memory DC that is compatible with our screen DC (ps.hdc)
             hdc2 = CreateCompatibleDC(ps.hdc)
             'save the memory dc handle
             #g_dat.dcHand = hdc2
            'create our memory bitmap that is compatible with our screen DC (ps.hdc) and make it 200 x 200 pixels
            'and save the PREVIOUS bitmap handle
             #g_dat.bitHand = SelectObject(hdc2, CreateCompatibleBitmap(ps.hdc, 200, 200))
          endif
          hdc2=#g_dat.dcHand
         'draw our control
```

Now, the first time WM_PAINT is sent after this instance of the control is created the memory DC and bitmap will be created and their handles are saved.

We also have to make changes to give the memory back to the system at the proper time. We need to change the following:

Code:
```
...
        EndPaint(hWnd, ps)
        DeleteObject(SelectObject(hdc2, hbmp))
        DeleteDC(hdc2)
        RETURN 0
```

by moving

Code:
```
        DeleteObject(SelectObject(hdc2, hbmp))
        DeleteDC(hdc2)
```

and modifying the lines to use the proper UDT elements so that the result appears as:

Code:
```
      case WM_DESTROY
          DeleteObject(SelectObject(#g_dat.dcHand, #g_dat.bitHand))
          DeleteDC(#g_dat.dcHand)
          FreeHeap(g_dat)
          return 0
```

Now, our memory DC and bitmap will be created once, when the first WM_PAINT message is sent after creation of the control, and they will be deleted once, when this instance of the control is being deleted.

There is another opportunity for optimization. With the code we have created so far we are setup to draw 100% of each instance of our control each time the WM_PAINT message is sent. But is that really necessary?

When you consider all the components of our control our drawing can be divided into two categories, static and dynamic.
The dynamic portion is the pointers. They are constantly changing position. The static portion contains everything else; the gage border, the dial with its numbering, the text labeling. If we draw the static portion once when an instance of the control is created, and each time its configuration is actually changed, we can save it and use it over and over without having to redraw it.

We can accomplish this by creating a second memory DC and bitmap to hold the static / background drawing. We get started by doing what we did above.

First, we add two new elements to the GAGEPARAMS UDT:

Code:
```
TYPE GAGEPARAMS
   UINT dcHand
   UINT bitHand
   UINT dcBack
   UINT bmBack
   INT          sStyle                  /*GAGE_STYLE*/
   STRING       sName                   /*GAGE_TITLE*/
   STRING       sMultiUnits             /*GAGE_UNITS*/
   INT          nRangeMinAct            /*GAGE_RAWRNG*/
   INT          nRangeMaxAct            /*     "     */
   INT          nRangeMinDial           /*GAGE_DIALRNG*/
   INT          nRangeMaxDial           /*     "      */
   INT          nDialMulti              /*GAGE_MULTI*/
   UINT         nPanelColor             /*GAGE_SETPANEL_COLOR*/
   UINT         nGageColor              /*GAGE_DIAL_DARK*/
   INT          clockoffset             /*GAGE_CLKOFFSET*/
   INT          nPos1                   /*GAGE_SETPOS1*/
   INT          nPos2                   /*GAGE_SETPOS2*/
   FLOAT        nAngleStart
   FLOAT        nAngleEnd
ENDTYPE
```

Next we'll initialize the elements in the upper portion of the WM_CREATE section of our message handler.

Code:
```
               CASE WM_CREATE
                       g_dat = NEW GAGEPARAMS
                       SetWindowLong(hWnd, GWL_USERDATA, g_dat)
                       'init g_dat
                       #g_dat.dcHand                   = 0
                       #g_dat.bitHand                  = 0
                       #g_dat.dcBack                   = 0
```

```
                        #g_dat.bmBack                   = 0
                        #g_dat.sStyle                   = 0
                        #g_dat.sName                    = ""
                        #g_dat.sMultiUnits              = ""
                        #g_dat.nRangeMinAct             = 0
                        #g_dat.nRangeMaxAct             = 100
                        #g_dat.nRangeMinDial            = -1
                        #g_dat.nRangeMaxDial            = 14
                        #g_dat.nDialMulti               = 10
                        int r1,g1,b1
                        GageunRGBLM(GetSysColor(0), r1, g1, b1)
                        #g_dat.nPanelColor              = RGBA(r1,g1,b1,255)
                        #g_dat.nGageColor               = RGBA(255,255,255,255)
                        #g_dat.clockoffset              = 0
                        #g_dat.nAngleStart              = -230
                        #g_dat.nAngleEnd                = 50
                        RETURN 0
```

The following shows the modification to the WM_PAINT handler where we create the second DC and bitmap and store their handles:

Code:

```
      CASE WM_PAINT
         BeginPaint(hWnd, ps)
          if #g_dat.dcHand = 0
            'create our memory DC that is compatible with our screen DC (ps.hdc)
             hdc2 = CreateCompatibleDC(ps.hdc)
             'save the memory dc handle
             #g_dat.dcHand = hdc2
            'create our memory bitmap that is compatible with our screen DC (ps.hdc) and make it 200 x 200 pixels
            'and save the PREVIOUS bitmap handle
             #g_dat.bitHand = SelectObject(hdc2, CreateCompatibleBitmap(ps.hdc, 200, 200))
          endif
          hdc2=#g_dat.dcHand

          if (!#g_dat.dcBack)
             'create our bg memory DC that is compatible with our screen DC (ps.hdc)
             #g_dat.dcBack = CreateCompatibleDC(ps.hdc)
            'create our bg memory bitmap that is compatible with our screen DC (ps.hdc) and make it 200 x 200 pixels
             #g_dat.bmBack = SelectObject(#g_dat.dcBack, CreateCompatibleBitmap(ps.hdc, 200, 200))
            'call our subroutine to draw the background of this instance of the control
             GageDrawbackground(g_dat)
          endif
         'draw our control
```

Notice the addition of

Code:

```
            GageDrawbackground(g_dat)
```

We have to split the actual drawing commands in to because they will be drawn at different times. We will cover that subroutine shortly.

To finish up with what we did previously we need to give the memory back when we are through with it. We add that code here:

Code:

```
      case WM_DESTROY
        DeleteObject(SelectObject(#g_dat.dcHand, #g_dat.bitHand))
        DeleteDC(#g_dat.dcHand)
        DeleteObject(SelectObject(#g_dat.dcBack, #g_dat.bmBack))
        DeleteDC(#g_dat.dcBack)
        FreeHeap(g_dat)
        return 0
```

We now need a way to copy the background that we have drawn into the DC that we're going to draw to gage pointers in. There's a simple API function to accomplish that:

Code:

```
        BitBlt(hdc2, 0, 0, 200, 200, #g_dat.dcBack, 0, 0, SRCCOPY)
```

The placement of this function is very important. It has to be after the background is drawn and before we draw any of the gage pointers. The following shows the proper location:

Code:

```
      CASE WM_PAINT
         BeginPaint(hWnd, ps)
          if #g_dat.dcHand = 0
            'create our memory DC that is compatible with our screen DC (ps.hdc)
```

```
            hdc2 = CreateCompatibleDC(ps.hdc)
            'save the memory dc handle
            #g_dat.dcHand = hdc2
           'create our memory bitmap that is compatible with our screen DC (ps.hdc) and make it 200 x 200 pixels
           'and save the PREVIOUS bitmap handle
            #g_dat.bitHand = SelectObject(hdc2, CreateCompatibleBitmap(ps.hdc, 200, 200))
         endif
         hdc2=#g_dat.dcHand

         if (!#g_dat.dcBack)
            'create our bg memory DC that is compatible with our screen DC (ps.hdc)
            #g_dat.dcBack = CreateCompatibleDC(ps.hdc)
           'create our bg memory bitmap that is compatible with our screen DC (ps.hdc) and make it 200 x 200 pixels
            #g_dat.bmBack = SelectObject(#g_dat.dcBack, CreateCompatibleBitmap(ps.hdc, 200, 200))
           'call our subroutine to draw the background of this instance of the control
            GageDrawbackground(g_dat)
         endif
         BitBlt(hdc2, 0, 0, 200, 200, #g_dat.dcBack, 0, 0, SRCCOPY)
        'draw our control
```

Our code now is at the point where, when the first WM_PAINT message for an instance of a control is sent, our background DC is created and the subroutine to draw the background is called. After that first time it is not called again.

There is a problem with that. Our command to create an instance of a control causes the WM_CREATE message to be sent followed by a WM_PAINT message. This happens before we have used our configure command. With the current code code we draw the background only one time. We need to make it redraw when we change its configuration.

We need to go back and look at our configure subroutine that we covered in an earlier section. The code is in Section 6 of CCT_lib.iwb, shown below:

Code:

```
SUB ConfigGageRLM(win as WINDOW,_
                  style as INT,_
                  title as STRING,_
                  u as STRING,_
                  rawmin as INT,_
                  rawmax as INT,_
                  dialmin as INT,_
                  dialmax as INT,_
                  sf as INT,_
                  id as UINT )
   SENDMESSAGE(win, GAGE_STYLE, 0,style, id)
   STRING ttemp=title
   select style
      case ROUND270B
      case& ROUND270T
      case& ROUND270R
      case& ROUND270L
         if (dialmax-dialmin < 1) | (dialmax-dialmin > 15)
            dialmin=0 : dialmax=1
            ttemp="Dial Min/Max Error!"
         endif
   endselect
   SENDMESSAGE(win, GAGE_TITLE, 0,ttemp, id)
   SENDMESSAGE(win, GAGE_UNITS, 0,u, id)
   SELECT style
      CASE ROUNDCLOCK12
      CASE& ROUNDCLOCK24
         SENDMESSAGE(win, GAGE_CLKOFFSET, dialmin,0, id)
      DEFAULT
         SENDMESSAGE(win, GAGE_RAWRNG, rawmin,rawmax, id)
   ENDSELECT
   SELECT style
      CASE ROUND360SGL
      CASE& ROUND360DBL
      CASE& ROUND2PENV
      CASE& ROUND2PENH
         dialmin = 0 : dialmax = 10
   ENDSELECT
   SENDMESSAGE(win, GAGE_DIALRNG, dialmin,dialmax, id)
   SENDMESSAGE(win, GAGE_MULTI, 0,sf, id)
   GageInvalidateBackground(win, id)
   RETURN
ENDSUB
```

The last line of code before the RETURN statement is what we are interested in. We said at the time we would cover it later and and we created the following subroutine shell in Section 8 of CCT_lib.iwb.

Code:

```
SUB GageInvalidateBackground(win as WINDOW, id as UINT)

ENDSUB
```

The purpose of this routine is force the background to be redrawn whenever we change an instance of a control's configuration.
the first thing we need to do is get a copy of the parameters that are stored for this instance with the following two lines of code:

Code:
```
        pointer g_dat = GetWindowLong(GETCONTROLHANDLE(win,id), GWL_USERDATA)
        settype g_dat, GAGEPARAMS
```

The first line we use to get the location of our GAGEPARAMS UDT structure for this instance of the control. Since the definition of the type of variable a pointer points to can not cross module/subroutine lines we have to add the second to define the type. It allows us to use

Code:
```
g_dat.element
```

instead of

Code:
```
#<GAGEPARAMS>g_dat.element
```

in our code.

We now add the following:

Code:
```
        if (g_dat && g_dat.dcBack)
                DeleteObject(SelectObject(g_dat.dcBack, g_dat.bmBack))
                DeleteDC(g_dat.dcBack)
                g_dat.dcBack = 0
                g_dat.bmBack = 0
        endif
```

If the background DC exists we delete the bg bitmap and DC. We then set the two associated elements to zero.

We know, from our discussion above that if a message is sent to the control's handler that the background DC and bitmap will be created if g_dat.dcBack is equal to zero. And we just got through setting it to zero. So, all we are missing is a way to make it redraw. We will accomplish it by using the following OS API function.

Code:
```
        InvalidateRect(GETCONTROLHANDLE(win,id), NULL, FALSE)
```

The above line will cause a WM_PAINT message to be sent which will result in the background, with the new configuration parameters, being redrawn.

Our complete subroutine will appear as follows in Section 8 of CCT_lib.iwb.

Code:
```
SUB GageInvalidateBackground(win as WINDOW, id as UINT)
        pointer g_dat = GetWindowLong(GETCONTROLHANDLE(win,id), GWL_USERDATA)
        settype g_dat, GAGEPARAMS
        if (g_dat && g_dat.dcBack)
                DeleteObject(SelectObject(g_dat.dcBack, g_dat.bmBack))
                DeleteDC(g_dat.dcBack)
                g_dat.dcBack = 0
                g_dat.bmBack = 0
        endif
        InvalidateRect(GETCONTROLHANDLE(win,id), NULL, FALSE)
ENDSUB
```

As promised earlier, the last item we need to address is our background drawing routine. The skeleton for it is as follows:

Code:

```
SUB GageDrawbackground(pointer g_dat)
        settype g_dat,GAGEPARAMS
        uint hdc2 = #g_dat.dcBack
       'draw background


ENDSUB
```

As in other cases we have to specify the type for our g_dat pointer. And we create a variable to hold the handle to the DC we will be drawing to.

At this point our structure is established so that we can start drawing with GDI API functions.

_____

Coming Next - GDI+ Setup

IonicWind Software

IWBasic => Create a Custom Control => Topic started by: LarryMc on September 10, 2011, 03:41:53 PM

Title: 18. GDI+ Setup
Post by: LarryMc on September 10, 2011, 03:41:53 PM

As stated in the previous section, we can start drawing our control with the code structure already established. However, in the announcement of this tutorial effort we said we would be using the GDI+ library for our drawing.

Our main reason for opting for the GDI+ drawing functions is the lack of antialiasing with the standard GDI library.

A quick review.

When we normally draw a line at an angle it appears to be a stair step. That is because pixels are rectangles. If our infinitely thin line passes through a pixel the entire pixel is painted. This technique of representing a line with a stair step is called *aliasing*; the stair step is an alias for the theoretical line.

GDI+ uses a drawing technique where the closer a pixel is to our theoretical line the more solid the color is. The further a pixel is from the line the more transparent it is. This type of rendering is called *antialiasing* and results in a line that the human eye perceives as more smooth. The attached image of two ellipses is an example from the OS SDK.

Antialiasing also helps the appearance of text that is rotated from the horizontal.

Since our gage control is round and the pointers rotate we will get the best appearance by using GDI+.

In order to use the GDI+ library functions we must first initialize GDI+. We only need to do this once per application. And, as usual, we will need to clean up when we are through.

We start by defining two global variables:

Code:
```
        GdiplusStartupInput     GDI_Start
        UINT                    GDI_Token
```

The GdiplusStartupInput UDT structure is really of no concern to us except for one element which we will use to tell us whether or not we have already initialized GDI+.

The GDI_Token variable, which will have its value set when GDI+ is initialized, will be used during our cleanup.

We'll place the above two lines of code at the bottom of Section 2 in CCT_lib.iwb.

We then add the following code to the WM_CREATE portion of our message handler routine:

Code:
```
        CASE WM_CREATE
```

```
          ...
          IF (!GDI_Start.GdiplusVersion)
               GDI_Start.GdiplusVersion = 1
               IF GdiplusStartup(&GDI_Token, &GDI_Start, NULL)
                  MESSAGEBOX 0,"Error initializing GDI+/nCan not proceed.","Fatal Error"
                  END
               ENDIF
               ONEXIT &ReleaseGdip, 0
          ENDIF
          ...
```

When the first instance of our control is created in the user's application the IF statement will be TRUE. The next line will insure that the IF statement will be FALSE when additional instances of our control are created. This works because GDI_Start is a GLOBAL variable and not a local variable.

We then call the GdiplusStartup function. We pass it the addresses of our two global variables. If the function fails for any reason we show an error message and close the application.

If the function is successful we use the IWBasic ONEXIT function to add our cleanup subroutine to the internal list of routines to execute when an application is shutdown.

The following is our cleanup subroutine:

Code:

```
SUB ReleaseGdip(pointer p),INT
        GdiplusShutdown(GDI_Token)
        RETURN 0
ENDSUB
```

Passing of a pointer parameter is a requirement of the ONEXIT function. In this application we don't need to use it. So here, all we are doing is calling the OS GdiplusShutdown function. And we are passing it the value of our global GDI_Token variable.

The above subroutine will be placed at the bottom of Section 8 of CCT_lib.iwb. With that we have completed the required initialization and clean up of GDI+.

But we still can't use GDI+ drawing functions yet.

We spent the previous two sections discussing device contexts and how we draw to them. It only makes sense that we some how tie GDI+ to the two DCs we've already created code for.

We will need to add the following line right before we start our block of code that does the drawing:

Code:

```
   GdipCreateFromHDC(hdc2, &pGraphics)
```

This will add a GDI+ graphics object to our DC.

And as always we will need to do our cleanup by adding the following line to both.

Code:

```
if (pGraphics) then GdipDeleteGraphics(pGraphics)
```

Of course, in both places we will need to declare that pGraphics is a POINTER.

At this point we have completed all our preparations so that we can start using our

drawing functions.

_____

Coming Next - Drawing the Background

IonicWind Software

IWBasic => Create a Custom Control => Topic started by: LarryMc on September 10, 2011, 09:39:31 PM

Title: 19a. Drawing the Background
Post by: LarryMc on September 10, 2011, 09:39:31 PM

Part A

We're going to start by creating ten instances of our control right off the bat. Ten because that's how many different styles of gages we have. We're not going to configure them yet either.

So let's add the following code to Section 4 of CCT_test.iwb.

Code:
```
CreateGageRLM(main,10,30,200,1 )
CreateGageRLM(main,220,30,200,2 )
CreateGageRLM(main,430,30,200,3 )
CreateGageRLM(main,640,30,200,4 )
CreateGageRLM(main,10,270,200,5 )
CreateGageRLM(main,220,270,200,6 )
CreateGageRLM(main,430,270,200,7 )
CreateGageRLM(main,640,270,200,8 )
CreateGageRLM(main,10,510,200,9 )
CreateGageRLM(main,220,510,200,10 )
```

They will all have a radius of 200 pixels.

If we compile our project at this time the reader will see ten, 200 x 200 pixel, black rectangles. And that is with no actual drawing commands in our program. That is what the OS has done for us. This is showing us that at least each instance of the control is being created.

We'll now move our focus to the GageDrawbackground subroutine in Section 8 of CCT_lib.iwb.

The first command we're going to use is:

Code:
```
GdipSetSmoothingMode(pGraphics, SmoothingModeAntiAlias)
```

This will implement the antialias drawing mode for the pGraphics object.

We'll now clear the control's rectangle and fill it with the color stored in #g_dat.nPanelColor. Right now it is the default; a light grey. This is the code we'll add for that:

Code:
```
GdipGraphicsClear(pGraphics, #g_dat.nPanelColor)
```

Our GageDrawbackground subroutine now looks like this:

Code:
```
SUB GageDrawbackground(pointer g_dat)
        settype g_dat,GAGEPARAMS
        pointer pGraphics
        if (!GdipCreateFromHDC(#g_dat.dcBack, &pGraphics))
                uint hdc2 = #g_dat.dcBack
                'set the pGraphic object's drawing mode to antialiasing
```

```
                GdipSetSmoothingMode(pGraphics, SmoothingModeAntiAlias)
                'clear our contol's rectangle and fill it with the color stored in #g_dat.nPanelColor
                GdipGraphicsClear(pGraphics, #g_dat.nPanelColor)



                GdipDeleteGraphics(pGraphics)
        endif

ENDSUB
```

If the project is recompiled at this time the reader will see that all the rectangles have changed color. The reader can experiment with the color by going to the message handler WM_CREATE section and changing the number in parentheses for this line:

Code:

```
GageunRGBLM(GetSysColor(0), r1, g1, b1)
```

or entering reb,green,blue values directly in this line:

Code:

```
#g_dat.nPanelColor              = RGBA(r1,g1,b1,255)
```

Next we're going to draw the outer ring or frame of our gage. The color is a fixed, near black color. This is the code we will use:

Code:

```
                'draw outer gage ring
                FLOAT penwidth=8
                if (!GdipCreatePen1(RGBA(10,10,10,255),penwidth,UnitPixel,&pPen))
                        GdipDrawArc(pGraphics,pPen,5, 5, 190, 190,0,360)
                        GdipDeletePen(pPen)
                endif
```

First we define a variable to hold the width of the line we want to draw. We define it as a FLOAT because that is what the GDI+ function GdipCreatePen1 requires. The function also requires a RGBA color value, an OS constant that defines what units of measure the pen width is in, and a pointer variable that the function can store its pen information handle in. (That means we need to go up to the start of this routine and define the pPen pointer).

If the function is successful it will return 0 which makes the IF statement TRUE. In that case we will use the GdipDrawArc function to draw the circle. The parameters that are passed are the handle to the pGraphics object we are drawing to, the handle to the pPen we are going to draw with, the upper left corner coordinates of the rectangle that will contain our arc, the height and width of the rectangle that will contain the arc, the starting point of the arc in degrees and the ending point of the arc in degrees.

Following that function is our cleanup; we delete the pPen and return its memory to the system.

Next we're going to draw the center of our gage with the following code:

Code:

```
                'fill center of gage
                if (!GdipCreateSolidFill(#g_dat.nGageColor,&pBrush))
                        GdipFillEllipse(pGraphics,pBrush, 9, 9, 182, 182)
                        GdipDeleteBrush(pBrush)
                endif
```

With this code we're going to create a solid fill brush the same way we created the pen above. The color of the brush will be the value contained in #g_dat.nGageColor. The GdipFillEllipse function parameters are the handle to the pGraphics object we are drawing to, the handle to the pBrush we are going to paint with, the upper left corner coordinates of the

rectangle that will contain our ellipse, and the height and width of the rectangle that will contain the ellipse. (And, as before, we'll have to go to the top of the of the subroutine and declare pBrush as a POINTER.) And finally, when we're through with the brush we have to do our cleanup and delete it.

The following is how the GageDrawbackground subroutine now appears:

Code:

```
SUB GageDrawbackground(pointer g_dat)
        settype g_dat,GAGEPARAMS
        pointer pGraphics, pPen, pBrush
        if (!GdipCreateFromHDC(#g_dat.dcBack, &pGraphics))
                uint hdc2 = #g_dat.dcBack
                'set the pGraphic object's drawing mode to antialiasing
                GdipSetSmoothingMode(pGraphics, SmoothingModeAntiAlias)
                'clear our contol's rectangle and fill it with the color stored in #g_dat.nPanelColor
                GdipGraphicsClear(pGraphics, #g_dat.nPanelColor)

                'draw outer gage ring
                int penwidth=8
                if (!GdipCreatePen1(RGBA(10,10,10,255),penwidth,UnitPixel,&pPen))
                        GdipDrawArc(pGraphics,pPen,5, 5, 190, 190,0,360)
                        GdipDeletePen(pPen)
                endif

                'fill center of gage
                if (!GdipCreateSolidFill(#g_dat.nGageColor,&pBrush))
                        GdipFillEllipse(pGraphics,pBrush, 9, 9, 182, 182)
                        GdipDeleteBrush(pBrush)
                endif


                GdipDeleteGraphics(pGraphics)
        endif

ENDSUB
```

Coming Next - Drawing the Background - Part B