Custom Controls in a Box (-
lbjoseph)

Making your own custom control with a graphicbox in Liberty BASIC.

Throughout the ages, mankind has sought after customization and personalization. Now, with the power of Liberty BASIC 4, you can create your very own custom control inside of a simple graphicbox. The following article expects you to be familiar with the Liberty BASIC language, as well as with Liberty BASIC's native drawing commands.

In this article, we will discuss the techniques that I (-
lbjoseph) think are necessary to creating a good custom control.

Code for this article is on a separate page.

Table of Contents

Why a custom control?

Because you can. But there's more to it than that - by creating your own custom control, its look and feel will stay consistent throughout all the various versions of Windows. Secondly, you can make them much easier to use than the Windows API controls. It is also possible to setup your custom control's functions/subs without global variables (depending on what kind of control you make). Another advantage of custom controls is that they do not require a huge wrapper of subs and functions to deal with the Windows API calls, callbacks, messages, and so on.

The container

First of all, we need some sort of object that we can manipulate to turn into our own object. The Liberty BASIC graphicbox is just the thing. Not only does it let you draw on it with LB's native drawing commands, but it lets you set event handlers to branch labels or subs when it is clicked, etc. This allows you to make a click-able control that looks beautiful to the eyes (some sarcasm implied).

The foundation of what we want

In this article we'll walk through custom control creation by creating a "fake" hyperlink control. There's are a couple of things we want specifically from this control:

1. It needs to have a roll-over effect (becomes underlined upon mouse-over)
2. We want to be able to create as many as we like
3. And we want it to specify its properties upon creation - like background color, text color, font, etc.

So, let's go ahead and create the basis of our program. We'll need a window with a graphicbox in place and a trapclose event setup so our program will actually close when someone hits the little red x.

If we wanted to have more than one hyperlink, we'd make a graphicbox for every link we want. More on that later.
Click here for code.

If you run the code above, things don't look too promising. However, this is just the foundation of our program.

There are a couple of things you need to note about the above code:

First, we issued a "Stylebits" Command - we used the flag _WS_BORDER in the removeBits section. This removes the border of the graphicbox. This command is necessary to remove clutter which could mess up the way our control looks (especially in a dialog window).

Secondly, we created two "stubs" - [Link1.Click] and [Link1.Move] - these are event handlers for

graphicbox. We'll find out what these are for shortly.

Now we have the foundation of a custom control - a graphicbox without a clunky border and two stubs for when the user clicks the graphicbox, and moves the mouse over the graphicbox.

Data management

Here's where the fun begins. We know we need several parameters to create a hyperlink:

1. We need the handle to the "parent" graphicbox container. We'll call this variable "gbHndl$". In our case, the programmer would pass the value "#Win.Link1" into the function.
2. We need the width of the hyperlink - we'll call this "width"
3. We need the height of the hyperlink - we'll call this "height"
4. We need the text for the hyperlink - we'll call this "text$"
5. We need to know what font to use - we'll call this "font$"
6. We need to know the background color of the window the link is on (so it won't look ugly) - we'll call this "backcolor$"
7. We need to know what color to make the hyperlink -we'll call this "linkcolor$"
8. We need the name of the sub or branch label that contains the user's code to execute upon clicking the link. We'll call this "eventClick$". In our case, the programmer would pass the value "[Link1.Click]" into the function.
9. We'll need the name of the sub or branch label that contains the user's code to execute when the mouse moves over the link. We'll call this "eventMove$"
10. We need the height of the font the user specified for the hyperlink - we'll call this "fontHeight"
11. The hyperlink also needs to know what state it is in - whether or not is "active". We'll call this "hlinkActive".

We keep track of 11 variables for the hyperlink. We'll need to remember these so we can render the hyperlink. To avoid flickering the graphicbox, we'll only render the hyperlink when we absolutely need to. By calculating to see if the mouse is inside the coordinates of the text, we can find out whether or not to underline the hyperlink. If so, we'll redraw the link and change the value of hlinkActive to 1 (if the mouse is over the text) or 0 (if the mouse is not over the text).

The guts of our hyperlink program

Here's the updated program (heavily commented) with the sub and functions required for the hyperlink:
Click here for code.

What does it all mean?

To make the most out of this, you should probably look over the code and read the comments. It's designed to be easy to understand. You should be able to keep up quite well after doing so.

The function cCc.Hyperlink$ (the cCc stands for create custom control) is called after the window has been setup like this:
Click here for code.

The arguments passed into this function correspond with the parameters (representational variables) received by the function cCc.Hyperlink$():
Click here for code.

The function returns a string of values put together. When we called this function, we assigned the variable Link1$ to be that string.

Earlier, we mentioned that we need an event for when the mouse moves over the link. If you compare the two above, you'll notice that we specified the branch label [Link1.Move] to be the block of code that Liberty jumps to when the mouse moves over the graphicbox #Win.Link1. Let's take a look at that block:
Click here for code.

Notice that we called the sub in our program. This sub, cC.RollHyperlink (cC stands for Custom Control), checks to see if the mouse is directly over the rendered text (using the graphics commands, and a little bit of math and conditional checking). The sub then draws the hyperlink either in its active (underlined) form or inactive (not underlined) form depending on if the mouse is over it or not.

In the sub, we passed the value Link1$ as the handle to our link. It contains the string of information returned by the cCc.Hyperlink$() function. The sub breaks this information down, and uses it to see if it needs to redraw the hyperlink. If it does, it changes the value of the paremeter hlinkActive, which happens to be second word inside Link1$.

When the sub is done, it puts the link handle back together:
Click here for code.

Now, remember that in our case, we passed the value Link1$ in the sub. If you were to compare the values of the string variable Link1$ when the mouse is over, and when it is not, you would get this:
Click here for code.

The first line is the value of Link1$ when the mouse is over the link. The second line is the value when the mouse is not over. There is only one difference between the two, and that is found in the fourth word. If you looked at the code, you'd realize that the fourth word in the "handle" of our link is the value of hlinkActive. But how are we able to change the actual value of a handle from the sub?

We take advantage of passing an argument by reference. Look at this:
Click here for code.

Notice the little "byref" that is thrown in right before Link$ (the parameter that represents the handle of the link passed in). Even though it doesn't turn blue in the Liberty BASIC 4.03 editor, it is still a recognized specification by Liberty BASIC's standards. Whenever you pass a variable into a sub or function by reference, the sub or function can actually modify the value of the variable you gave it! Isn't that amazing? Ok, well maybe it doesn't blow you out of your seat, but it can be really useful - as we just saw.

For more info, check out BYREF in the help file.

Utilizing the hyperlink

Ok, so we have this great hyperlink custom control, but how do we use it? Well, it's really quite simple to use. All you have to do is look a little closer at the demo program.

You already know how to make a hyperlink. Just assign a value like we did in the demo to the result of cCc.Hyperlink$(), with your own arguments.
Click here for code.

As you know, the hyperlink function requires two events. The second event, what to do on mouse over, requires us to call the sub cC.RollHyperlink.

So, once you have your events setup, all you have to do is call the cC.RollHyperlink sub from your mouse over event:
Click here for code.

Then, all you have to do is pass the handle to the link - in this case, it's Link1$.

When you use a branch label event handler (like the example above) you must use the special variables MouseX and MouseY for the mouse coordinates. Liberty BASIC always assigns the x and y position of the mouse into these special global variables.

What if you chose to use a sub for an event handler? You would have to make the handle to your link global:
Click here for code.

And your sub might look something like this:
Click here for code.

Notice the sub must accept three parameters. When Liberty BASIC calls a sub event for a graphicbox, it always passes the graphicbox handle, mouse x position, and mouse y position.

As for the mouse click event, you can just put your own code in a sub or branch label. If you use a sub, be sure to put a receiver for the graphicbox handle, mouse x, and mouse y coordinates!

Conclusion and a Progress bar control

Well, that's about it for the hyperlink. With the power of Liberty BASIC graphicboxes, you can create all kinds of custom controls.

An example of a progressbar custom control can be found below. It's really quite simple, if you understand the principles discussed in this article.
Click here for code.