HyperControl Sample Tutorial

Sample Description: <u>HyperControls</u>

Tutorial

HyperControls

In this tutorial you will learn how to create an **EditCombo** HyperControl. A HyperControl is a combination of   individual controls, such as Buttons, TextBoxes, and ComboBoxes, that behave as a single control and can then be reused on other forms.

The EditCombo HyperControl you will be creating looks and functions like the controls located at the top of the Property Editor. It will consist of (2) Buttons and (1) ComboBox. You will be able to use these controls as a single unit for editing property information. You will be able to enter a value in the ComboBox by typing or selecting from the list (if more than one value exists) and then accept or refuse this input by using the **OK** and **X** buttons.

In the process of building the HyperControl, you will learn how to:

• Build and use a *Procedural Property*

• Add a property to an object

• Add a method to an object

• Define a **Resize** event handler

• Initialize a control with a **Setup** event handler

Completed HyperControl

Creating and Abstracting the HyperControl

Begin by creating a blank form on which to test the HyperControl you are about to make. This form will become a test harness for testing the EditCombo. Then, draw a HyperControl on the form, abstract it, and draw the controls on the abstract. Since the purpose of the sample is to create a reusable EditCombo, the form is being used merely to test the HyperControl you are making.

HyperControl

To create a new Form:
- Click the New Form button on the main menu.

To create and abstract the HyperControl:
1. On the Controls palette, click the HyperControl icon and drag it on the form in the approximate size shown in the example.

2. While the control is still selected, abstract the HyperControl by clicking the Abstract button on the Main menu. Type in the name **EditCombo**.

The EditCombo abstract appears on the screen. This is the *parent* EditCombo object and is what you'll be drawing your controls on. The EditCombo that you see on the test form is a derived copy, or "child" and will inherit all property, method, and event attributes from the parent. It is automatically renamed **EditCombo1** since it is an instance of the parent object. To see this relationship as you add controls, move the parent EditCombo near the test form.

Note: Notice how abstracting a HyperControl is different than abstracting other controls. When abstracting other controls, the abstracted control does not appear visually on your screen as it does for the HyperControl. This is because you do not change the appearance of the other controls, you just change the behaviors on it, but you must change the appearance of the hypercontrol to define it.

Drawing Controls On the HyperControl

The next step is to draw the two buttons and the combobox on the abstracted HyperControl (EditCombo) and set the designated properties displayed below. The controls you will add in this section are shown as follows:

Button   **ComboBox**

To draw the controls:
1. Draw the first two Buttons displayed on the hypercontrol in the approximate size shown.

2. Next draw the ComboBox beside the buttons in the approximate size as shown.

3. To help align the buttons, make sure the height of the buttons matches the height of the combo by setting the Height property for all three controls to the same value. For example, if the height of the combobox is 360, the height of the two buttons should also be 360.

4. Because the buttons are square, make sure the Width property is set to the same value as the Height.

5. Once the buttons are sized and aligned correctly, set the following properties:

Control	Property	Value
Button1	Caption	X
	Name	BtnDelete
Button2	Caption	ok
	Name	BtnOK
ComboBox	Name	EditBox

Notice as you draw these controls on the EditCombo parent that they appear automatically on the EditCombo1 abstract. The EditCombo1 child on the form is inheriting these controls from the EditCombo parent.

Setting the Resize Method

After carefully aligning and sizing the controls, you will define a **Resize** method for the EditCombo so that the individual Button and ComboBox controls are accurately resized when the size of the EditCombo itself is changed. Currently the Resize method is defined on the HyperControl. Since you want to change the behavior of only the EditCombo and not the behavior of all HyperControls, you will need to *override* this method and define the new resize behavior on the EditCombo.

To set the Resize method:

1. Finger the EditCombo.

2. In the Methods Editor, select the Resize method.

3. Click the Override button in the Method Editor.

4. In the code area, change the current behavior using the following code:

```
Sub Resize()
  BtnDelete.Move(0, 0, BtnDelete.Width, BtnDelete.Height)
  BtnOK.Move(BtnDelete.Width, 0, BtnDelete.Width,  BtnDelete.Height)
  EditBox.Move((BtnDelete.Width * 2), 0, ScaleWidth - (2 *
  BtnDelete.Width), BtnDelete.Height)
  Height = BtnDelete.Height + (Height - ScaleHeight)
 End Sub
```

5. To check and add the method, click the Check Current Method button.

To test this resize method, try sizing the EditCombo1 on your form. The controls should remain aligned and sized correctly. You will notice that you can only adjust the width on the control, but not the height.

Defining a KeyUp Event Handler

Now that you've created the EditCombo HyperControl, you need to start defining behaviors for these controls. First, you need create the behavior for the combobox that makes typing and pressing return in the combobox mean the same as clicking **OK** directly. To do this you need to define an event handler on the **KeyUp** Event.

To define the KeyUp event handler:

1. Make sure the EditBox is still selected for editing, if not Finger it.

2. In the Methods Editor, select KeyUp from the Events combobox.

3. In the code area, enter the following text between the Sub and End sub as shown:

```
Sub EditBox_KeyUp(KeyCode As Integer, ByVal shift As Integer)
  ' If the return key is struck, enter the value
  If keyCode = VK_RETURN && BtnOK.Enabled Then BtnOK_Click
End Sub
```

4. To check and add the handler, click the **Check Current Method** button.

Notice that the VK_RETURN constant is used in the code. This constant represents a carriage return as a number, which is currently 13. By using constants you can easily update values such as this throughout your code if changes are made.

Creating a Procedural Property on EditCombo

To further define the behavior of the EditCombo, you will need to define a **Value** property which will allow the control to bring selected values into the ComboBox. This property is a *Procedural Property* that allows you to filter and set items such as values automatically. It is added a little differently than other properties.

To create a procedural property:

1. Make sure the EditCombo is still selected for editing, if not Finger it.

2. In the Methods Editor, click the **Add Method** Button until the Property template appears as shown:

```
Property <name> Get <getName> [Set <setName>] [As <type>]
```

3. Enter the following code into the template as shown below:

```
Property Value Get getValue Set setValue As String
```

4. To check and add the Value property, click the **Check Current Method** Button.

Value appears in the methods combo list and in the property list. To complete the definition for this property, you must also define two procedures. The procedures you must define are the **getValue** and the **setValue** procedures. These procedures "filter" the information that can go into the combobox.

Defining the Procedures for the Value Property

Since there is currently no value available for the Value property you just added, you must add a function to get this value. The function you must define to get the value for the Value property is the getValue function.

To add the getValue function:

1. In the Methods Editor, click the Add Method button until the Function template appears.

2. Enter the following:

```
Function getValue() As String
  ' Store the string in the hypercontrol's Caption property
  getValue = Caption
End Function
```

3. To check and add the getValue function, click the **Check Current Method** Button.

The return value of the getValue procedure is immediately picked up by Value. Currently this value is *read only*. To change this, you must define a setValue procedure as well.

To add the setValue sub:

1. In the Methods Editor, click the Add Method button until the Sub template appears.

2. Enter the following:

```
Sub setValue(entry As String)
  ' Store the string in the hypercontrol's Caption property
  Caption = entry
  EditBox.Text = entry
  BtnOK.Enabled = True
End Sub
```

3. To check and add the setValue sub, click the **Check Current Method** Button.

> **Note:** Notice that the value is stored in the EditCombo's Caption property. The Caption property is a good location for storage since it can store the string, yet the caption is not visible on the form. The value is stored in this location until it is either rejected or accepted by clicking the ok or delete (X) buttons.

The last part of the setValue procedure enables the **OK** button when text is entered into the combobox.

To test your work, use the Property Editor to enter a value into the Value property. As your entry is entered, the setValue method is automatically executed.

Defining the Click Event Handler for the X button

Now that you've defined the behavior of the combobox, you need to define the behavior for the buttons. Begin by defining the behavior for the delete button. When the button named BtnDelete is clicked, it prevents the information from updating in the Methods Editor when clicked. To do this you must define a click event handler for BtnDelete.

To define the BtnDelete click event handler:

1. Finger BtnDelete (X button).

2. In the Methods Editor, select the Click event.

3. Enter the following code between the Sub and End Sub as shown:

```
Sub BtnDelete_Click()

  If EditBox.Text = Value Then

    EditBox.Text = ""

  Else

    EditBox.Text = Value

  End If

  BtnOK.Enabled = False

End Sub
```

4. To check and add the setValue sub, click the **Check Current Method** Button.

To test this, in the Property Editor type a value for the Value property, then click the Delete (**X**) Button.

Defining the Click Event Handler for the OK Button

Next define the click event handler for the OK button. When you click OK, you want the click to run the setValue procedure.

To define the BtnOK click event handler:

1. Finger the ok button.

2. In the Methods Editor, select the Click event.

3. Enter the following code between the Sub and End Sub as shown:

```
Sub BtnOK_Click()
   ' The EditBox changes to the Value property
   Value=EditBox.Text
   SendEvent Click
End Sub
```

4. To check and add this method, click the **Check Current Method** Button.

In the above code, the SendEvent Click performs the click if the carriage return is received. This event is sent to the other controls informing them that input has been accepted.

To test this functionality, type something in the actual combo, click ok and see that it is actually set.

Publishing ComboBox Methods on the Hypercontrol

To utilize the ComboBox on the HyperControl, you must define an **AddItem** and a **Clear** method. The AddItem method will place items in a dropdown list when there is more than one available value. The Clear method will remove the items from the list.

To define the AddItem method:
1. Finger EditCombo.

2. In the Methods Editor, click the **Add Method** Button until the Function template appears.

3. Enter the following code between the Function and End Function as shown:

```
Function AddItem(ByVal item As STring) As Integer
   'Exposing ComboBox's AddItem method through the hypercontrol.
   AddItem = EditBox.AddItem(item)
End Function
```

4. To check and add the method, click the **Check Current Method** Button.

To define the Clear method:
1. In the Methods Editor, click the Add Method button until the Sub template appears.

2. Enter the following code between the Sub and End Sub as shown:

```
Sub Clear ()
   ' Expose the ComboBox's Clear method through the hypercontrol
   EditBox. Clear
End Sub
```

3. To check and add the method, click the **Check Current Method** Button.

By doing this you have essentially published a known method on a parent in order to forward the functionality to one of the controls within the parent.

Initializing the HyperControl

As a final step, to initialize the control when it is dropped onto the form, you will define a **Setup** method.

To define the Setup method:

1. In the Methods Editor, click the **Add Method** Button until the Sub template appears.

2. Enter the following code between the Sub and End Sub as shown:

```
Sub Setup()
   'Clear out the value entry
   Value=""
End Sub
```

3. To check and add the method, click the **Check Current Method** Button.

Using the Test Harness

To test the functionality of the EditCombo, display the Samples Browser and load the **HyperControl** sample, which is located in the Concepts section.

Manipulate the controls to see how it works. Use the Property and Methods Editors to explore the EditCombo's functionality further.