

Programming Components with Microsoft Visual Basic 6.0

Chapter 6

Creating ActiveX Controls

Source : <http://www.visualbasicbooks.com>

جمع وتنسيق : صاعق

منتديات الفريق العربي للبرمجة

[Http://www.arabteam2000-forum.com](http://www.arabteam2000-forum.com)

Creating ActiveX Controls

A wide assortment of development tools—including Microsoft Visual C++, Microsoft Visual J++, Microsoft Visual Basic, Microsoft Visual FoxPro, Borland Delphi, PowerBuilder, and Microsoft Access—support ActiveX controls. Until recently, however, the only tool available for creating ActiveX controls was Visual C++. This relegated development of ActiveX controls to the province of those who knew lower-level programming languages such as C and C++. With Visual Basic, Microsoft has given developers another tool for creating ActiveX controls. Using Visual Basic, you can create controls for your own software projects or for distribution and sale to other developers for use in their software. In fact, a developer writing applications in C++ and Java can use the ActiveX controls you create in Visual Basic.

ActiveX controls that you build in Visual Basic (or any language, for that matter) are not only useful in development tools but also in end-user applications. For example, you can embed an ActiveX control created in Visual Basic in an HTML document that will be viewed in a Web browser such as Microsoft Internet Explorer. The application in which an ActiveX control is displayed is referred to as the control's *container*. Visual Basic is the ActiveX control container used for most of the examples in this chapter—however, keep in mind that your control might be activated in containers other than Visual Basic.

ActiveX controls are comparable to applets written in Java. An *applet* is any small program or utility, but of late, the term applet has come to mean a small application embedded in an HTML document that can be displayed by a Web browser. In fact, the Microsoft Java VM (virtual machine) exposes Java Bean components as ActiveX controls. The advantage to this is that applets can be thought of as ActiveX controls, which in turn means that they can be scripted using ActiveX Scripting languages such as VBScript or JScript.

What Makes ActiveX Controls Different from Standard EXE Projects?

Despite all the racket I made in previous pages about true programmers being felled by Visual Basic interlopers, creating an ActiveX control is (fortunately) difficult, even in Visual Basic. Controls are definitely one of the most complex project types to develop—so much so, that even experienced Visual Basic developers might have a hard time grasping the essence of creating ActiveX controls. Part of this complexity stems from the fact that controls are very different creatures from other types of software that you can build with Visual Basic.

For starters, having two personalities is beneficial when you are engaged in creating ActiveX controls in Visual Basic. ActiveX controls (and their developers) basically have two separate personalities. You are a Dr. Jekyll when you are developing a control in Visual Basic, sacrificing yourself on the altar of science. The developer using your component, however, sees you as a Mr. Hyde: a ruthless, self-serving murderer of innocent code. Once you comprehend both

facets of working with an ActiveX control, developing an ActiveX control in Visual Basic will be much easier.

Let's leave nineteenth-century literature and return to contemporary software. Think of a Standard EXE project. You develop the program in design mode and then test the program in run mode. Once you have the program working, you create an EXE file that always executes in run mode. Controls, likewise, are developed in design mode and are tested in run mode. However, since controls never run as stand-alone applications, you always need a test container in which to execute the control. The oddity is that once you have developed a control, produced an .OCX file, and given it to other developers to use in their software, the control is back to working in Visual Basic's design mode. This is the peculiar fact about the life of a control: even when Visual Basic is in design mode, controls are running.

ActiveX controls are comparable to applets written in Java. An *applet* is any small program or utility, but of late, the term applet has come to mean a small application embedded in an HTML document that can be displayed by a Web browser. In fact, the Microsoft Java VM (virtual machine) exposes Java Bean components as ActiveX controls. The advantage to this is that applets can be thought of as ActiveX controls, which in turn means that they can be scripted using ActiveX Scripting languages such as VBScript or JScript.

What Makes ActiveX Controls Different from Standard EXE Projects?

Despite all the racket I made in previous pages about true programmers being felled by Visual Basic interlopers, creating an ActiveX control is (fortunately) difficult, even in Visual Basic. Controls are definitely one of the most complex project types to develop—so much so, that even experienced Visual Basic developers might have a hard time grasping the essence of creating ActiveX controls. Part of this complexity stems from the fact that controls are very different creatures from other types of software that you can build with Visual Basic.

For starters, having two personalities is beneficial when you are engaged in creating ActiveX controls in Visual Basic. ActiveX controls (and their developers) basically have two separate personalities. You are a Dr. Jekyll when you are developing a control in Visual Basic, sacrificing yourself on the altar of science. The developer using your component, however, sees you as a Mr. Hyde: a ruthless, self-serving murderer of innocent code. Once you comprehend both facets of working with an ActiveX control, developing an ActiveX control in Visual Basic will be much easier.

Let's leave nineteenth-century literature and return to contemporary software. Think of a Standard EXE project. You develop the program in design mode and then test the program in run mode. Once you have the program working, you create an EXE file that always executes in run mode. Controls, likewise, are developed in design mode and are tested in run mode. However, since controls never run as stand-alone applications, you always need a test container in which to execute the control. The oddity is that once you have developed a control, produced an .OCX file, and given it to other developers to use in their software, the control is back to working in Visual Basic's design mode. This is the peculiar fact about the life of a control: even when Visual Basic is in design mode, controls are running.

Writing Your Own Control from Scratch

Writing a control from scratch affords you the greatest flexibility since you are not limited by any existing functionality. This allows you to do anything you want with your control's appearance and interface simply by putting code into the *Paint* event procedure used to draw your control. This is the model to select when creating a new visual widget such as a super-duper button.

Enhancing an Existing Control

Enhancing an existing control means using that control as a starting point and then adding your own properties, methods, and events. You have complete freedom in specifying the programmatic interface for your enhanced control. The constituent control's properties will be exposed only if you decide to expose them.

Modifying the appearance of an existing control is more difficult than creating a control from scratch, however, because the control you are enhancing already contains code to paint itself. Experienced Microsoft Windows programmers can use the *AddressOf* operator to make the constituent control a subclass so that it will notify you of its *Paint* events, but this is not an easy task. Using these hybrid controls is similar to what is done in the object-oriented concept of aggregation. Recall that aggregation occurs when you build a new object by accumulating objects that already exist.

Assembling a New Control from Several Existing Controls

The last control creation model expounded by Visual Basic is assembling a control from several existing controls. This model is similar to the second model in that they both utilize existing controls to build new controls.

The *UserControl* Object

An ActiveX control created in Visual Basic is always composed in a *UserControl* object, just as a Visual Basic application is always composed in a *Form* object. Like a Visual Basic *Form* object, a *UserControl* object has a code module and a visual designer (the *UserControl* designer), as shown in Figure 6-1. Note that you can also use class modules inside an ActiveX control project. In addition to a *UserControl* object, an ActiveX control in Visual Basic can have any other controls—called *constituent controls*—that you choose to place on the *UserControl* object. You use constituent controls only if you want to enhance an existing control or create a hybrid control consisting of several existing controls. You place constituent controls on the *UserControl* designer and set their properties in the same way you do when placing controls on a form.

When saved by Visual Basic, a *UserControl* object is stored in a plain text file with a .CTL extension. The .CTL file contains the source code and property settings for the *UserControl* object and for any constituent controls that might be present. If a *UserControl* object or its constituent controls use graphical elements, such as bitmaps, which can't be stored as plain text, Visual Basic stores those elements in a .CTX file with the same name as the .CTL file.

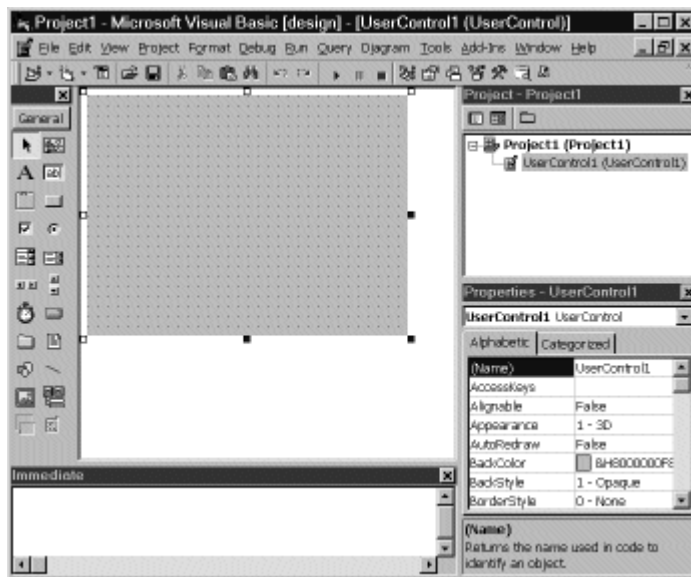


Figure 6-1. *The UserControl designer.*

The *UserControl* Object's Interface

If you have worked with Visual Basic for any length of time, you know that there are three basic members you use with controls: properties, methods, and events. Some items, however, are unique to *UserControl*. These are described in the following tables, which you can peruse to get an idea of the functionality available. We will go into detail on the important items later in this chapter and in successive chapters.

Note The following table includes properties that are not shown in the Properties window.

<i>UserControl Property</i>	<i>Property Description</i>
<i>AccessKeys</i>	String containing all mnemonics (hot keys) for this control.
<i>Alignable</i>	Determines whether the <i>UserControl</i> object can be aligned in its container.
<i>Ambient</i>	Returns a <i>UserControl</i> object that allows the programmer read-only access at run time to the ambient properties of the container, such as the <i>BackColor</i> property.
<i>CanGetFocus</i>	Determines whether the <i>UserControl</i> object can have the focus.
<i>ClipBehavior</i>	Indicates how a windowless <i>UserControl</i> 's appearance is clipped.
<i>ClipControls</i>	Determines whether graphics methods in <i>Paint</i> events repaint an entire object or only newly exposed areas.

<i>ContainedControls</i>	Returns the set of controls contained within the <i>UserControl</i> object.
<i>ControlContainer</i>	Determines whether the <i>UserControl</i> object can act as a container for other controls at design time.
<i>DataBindingBehavior</i>	Determines whether the <i>UserControl</i> object supports data binding.
<i>DataSourceBehavior</i>	Determines whether the <i>UserControl</i> object is a data source.
<i>DefaultCancel</i>	Determines whether the <i>UserControl</i> object behaves like a standard command button.
<i>EditAtDesignTime</i>	Determines whether the <i>UserControl</i> object will have an <i>Edit</i> verb on the control's menu that, if set to True, allows the control to become active at design time.
<i>EventsFrozen</i>	Indicates whether the container is ignoring events on the control.
<i>Extender</i>	Returns an object to the control that contains the values for the properties or the methods of the container.
<i>ForwardFocus</i>	Determines which control receives focus when an access key is pressed.

Note The following table includes properties that are not shown in the Properties window.

UserControl Property	Property Description
<i>HasDC</i>	Determines whether a unique display context is allocated for the control.
<i>HitBehavior</i>	Indicates which mode of automatic hit testing a windowless control employs.
<i>Hyperlink</i>	Returns a <i>Hyperlink</i> object used for browser-style navigation.
<i>InvisibleAtRuntime</i>	Determines whether the <i>UserControl</i> object is invisible at run time.
<i>ParentControls</i>	Returns the set of controls used by the <i>UserControl</i> object's container.
<i>PropertyPages</i>	Determines the type, number, and order of property pages

	for the <i>UserControl</i> object.
<i>Public</i>	Determines whether the control is accessible to other clients.
<i>RightToLeft</i>	Determines text display direction and visual appearance on a bidirectional system such as Arabic or Hebrew.
<i>ToolboxBitmap</i>	References a graphic that represents the <i>UserControl</i> object in the toolbox.
<i>Windowless</i>	Determines whether the <i>UserControl</i> object can be activated as windowless in a container that supports it.
UserControl Method	Method Description
<i>AsyncRead</i>	Asynchronously reads data from a URL or path.
<i>CancelAsyncRead</i>	Cancels an asynchronous read.
<i>CanPropertyChange</i>	Asks the container of the <i>UserControl</i> object whether a control property can be changed.
<i>DataMemberChanged</i>	Notifies clients that a data member of this data source has changed.
<i>PropertyChanged</i>	Signals that a property of the <i>UserControl</i> object has changed.
UserControl Event	Event Description
<i>AccessKeyPress</i>	Occurs when the user of the control presses one of the control's access keys, or when the Enter key is pressed after the developer has set the <i>Default</i> property to <i>True</i> , or when the Escape key is pressed after the developer has set the <i>Cancel</i> property to <i>True</i> . The author of the control enables the <i>Default</i> property and the <i>Cancel</i> property by setting the <i>DefaultCancel</i> property to <i>True</i> .
<i>AmbientChanged</i>	Occurs when the container of a <i>UserControl</i> object changes an ambient value.
<i>AsyncReadComplete</i>	Occurs when all of the data is available as a result of the <i>AsyncRead</i> method.
<i>AsyncReadProgress</i>	Occurs when more data is available as a result of the <i>AsyncReadProgress</i> method.
<i>EnterFocus, ExitFocus</i>	Occurs when the <i>UserControl</i> object is activated (<i>UIActivate</i>) or deactivated (<i>UIDeactivate</i>)—that is, receives or loses the focus.

<i>GetDataMember</i>	Occurs when a client is asking this data source for one of its data members.
<i>Show, Hide</i>	Occurs when the <i>UserControl</i> object is shown or hidden.
<i>HitTest</i>	Occurs in a windowless user control in response to mouse activity.
<i>InitProperties</i>	Occurs the first time a new instance of a <i>UserControl</i> object is created.
<i>ReadProperties</i>	Occurs when a <i>UserControl</i> object is asked to load an old instance in the persisted state from the <i>PropertyBag</i> object.
<i>WriteProperties</i>	Occurs when a <i>UserControl</i> object is asked to write an instance of a persistent state to a <i>PropertyBag</i> object.

Key Events in the Life of a *UserControl* Object

As we have mentioned before, you might develop an ActiveX control in Visual Basic at design time, and then another programmer might use this control to develop software at design time that in turn executes the control at run time. As a result, an ActiveX control needs to deal with many situations. As we mentioned early in this chapter, ActiveX controls are actually running most of the time that Visual Basic is in design mode. While the UserControl designer is open, an ActiveX control is in design mode. However, as soon as the UserControl designer closes, the control enters run mode automatically. The only indication of this transition to a running state is that the control's icon in the Toolbox becomes enabled.

The life of an ordinary form in a standard Visual Basic application is punctuated by certain key events, such as *Initialize*, *Load*, *QueryUnload*, and *Unload*. To create well-behaved software, you need to know at what stage in the form's life these events are fired. The same is true for controls. The key *UserControl* events are *Initialize*, *InitProperties*, *ReadProperties*, *WriteProperties*, and *Terminate*. Notice that ActiveX controls are missing *Load* and *Unload* events. These events are superseded by the *ReadProperties* and *WriteProperties* events. *Load* and *Unload* events don't make sense for controls since controls aren't loaded and unloaded like forms.

Why the Exercises Use Standard EXE Projects

In this chapter, Standard EXE projects are used for the exercises. This may seem odd since the focus of this chapter is ActiveX controls. To make the development of the exercises easier, ActiveX controls are added to Standard EXE projects. The controls work identically to those created in an ActiveX control project, with one exception: they are part of a standard .EXE file instead of being an .OCX file and are therefore private to the project. These controls can't be used by other software projects. This is a very easy problem to overcome. Once you have finished developing the control, simply create a new ActiveX control project and add the saved control (.CTL file) to it. Then you can build your control as an .OCX file. Visual Basic can have multiple projects open concurrently, and with this feature we could have built the controls as ActiveX control projects from the start.

I often find, however, that it is easy to get lost and confused with too many projects and files floating around. This is why we use Standard EXE projects instead.

Exercise 6-1

Let's create a test control that simply displays messages in response to the important events in this life of a control.

1. Open Visual Basic, and start a new Standard EXE project.
2. Choose Add User Control from the Project menu. On the New tab of the Add User Control dialog box, select the default User Control and click Open.
3. In the Properties window for the user control, set the *BorderStyle* property to *1 - Fixed Single*.
4. Close the UserControl designer.
5. Now place the new control on Form1.
6. In the Project window, double-click UserControl1 to reopen the UserControl designer.
7. Double-click the UserControl designer to open the Code window.
8. Code the following event procedures. You can use the Procedure box to select the event procedures and then add the message box code (shown in bold).

```
Private Sub UserControl_Initialize()  
MsgBox "Initialize"  
End Sub
```

```
Private Sub UserControl_InitProperties()  
MsgBox "InitProperties"  
End Sub
```

```
Private Sub UserControl_ReadProperties(PropBag As _  
PropertyBag)  
MsgBox "ReadProperties"  
End Sub
```

```
Private Sub UserControl_WriteProperties(PropBag As _  
PropertyBag)  
MsgBox "WriteProperties"  
End Sub
```

```
Private Sub UserControl_Terminate()  
MsgBox "Terminate"  
End Sub
```

```
Private Sub UserControl_Resize()  
MsgBox "Resize " & ScaleWidth & ", " & ScaleHeight  
End Sub
```

```
Private Sub UserControl_Show()  
MsgBox "Show"  
End Sub
```

9. Close the Code window, and then close the UserControl designer. Note that the control's icon in the toolbox has become enabled. This indicates that the control has been placed in run mode.

Note that the message boxes appear in the following order:

```
Initialize  
Resize 1524, 1284  
ReadProperties  
Show
```

(Your resize values will reflect the size of your control.)

10. Choose Save Project from the File menu. Save the files as LIFE.CTL, LIFE.FRM, and LIFE.VBP. Note that you must save a project with a control before you can run the project.
11. Choose Start from the Run menu. Notice the events fired:

```
Terminate  
Initialize  
Resize 1524, 1284  
ReadProperties  
Show
```

12. Choose End from the Run menu. Notice the events fired:

```
Initialize  
Resize 1524, 1284  
ReadProperties  
Show
```

13. When the message boxes become too annoying (they probably have already), go back into the UserControl designer's Code window and replace all the *MsgBox* statements with *Debug.Print*, which will direct all output to the Immediate window. Close the UserControl designer when you've finished.
14. Close the Form1 Form window. Notice the event fired in the Immediate window.

```
Terminate
```

15. In the Project window, double-click Form1 to open the Form window again. Notice the event fired in the Immediate window.

```
Initialize  
Resize 1764, 1164  
ReadProperties  
Show
```

16. Right-click on Form1's background (not on UserControl1), and select Update UserControls. Notice the string of events in the Immediate window.
17. The purpose of this exercise is to observe that ActiveX controls are alive from the moment the UserControl designer is closed. This action implicitly puts controls in the running state necessary to enable a developer to work with them. *UserControl* objects actually run in two modes: in Visual Basic's design mode when software is being developed, and as part of another program at run time. Many Visual Basic programmers (not you, gentle reader) put a control on a form and thereafter erroneously think of the control as a permanent fixture of the form. By observing the *Terminate* and *Initialize* events as they occur in this exercise, you can see that the

ActiveX controls are continually being created and destroyed in the Visual Basic environment.

Intrinsic Controls

Visual Basic comes with a bouquet of built-in controls. These are now called *intrinsic* controls to differentiate them from other types of controls, such as custom-built ones. The intrinsic controls are: *CheckBox*, *ComboBox*, *CommandButton*, *Data*, *DirListBox*, *DriveListBox*, *FileListBox*, *Frame*, *HScrollBar*, *Image*, *Label*, *Line*, *ListBox*, *OptionButton*, *PictureBox*, *Shape*, *TextBox*, *Timer*, and *VScrollBar*.

Properties

Now let's turn our attention to properties. Visual Basic has three basic categories of control properties: ambient, extender, and custom. *Ambient* properties provide your control with information about the state of its container. *Extender* properties are those that seem to be part of your control but that are actually provided at run time by the container. *Custom properties* are ones that you implement entirely on your own.

It might initially seem surprising that there is such a variety in types of properties. Most developers who write applications in Visual Basic think of the properties that appear in the Properties window as belonging to a control. In fact, other properties not displayed in the Properties window exist, and of the properties displayed, some do not belong to the control. Consider the *Top* and *Left* properties of a control. They describe where the control is situated within the container. These types of properties belong to the container rather than the control, even though they appear in the Properties window.

Ambient Properties

The container provides ambient properties, and you can think of these properties as providing your control with information regarding its current environment. In fact, ambient properties are exposed by the client site in which the control is embedded, and they reflect values that the container wants to communicate to the control so that the control will take on the characteristics of the form in which it lives. These are, effectively, hints about how the control can best display itself in the container. Think of a chameleon that camouflages itself by changing color to match its surroundings. The purpose of ambient properties is to help your control camouflage that it is not really a native part of its container.

For example, the ambient property *BackColor* informs a control of the background color of its container. This can then be used to set the control's own background color to blend better with the container. Ambient properties don't need to be obeyed strictly, but it is polite to do so when it makes sense. Visual Basic makes ambient properties available to your control through an *AmbientProperties* object, a reference to which is available in the *Ambient* property of the *UserControl* object. This leads to this syntax for accessing ambient properties:

```
UserControl.Ambient.AmbientPropertyName
```

Ambient properties are properties of the *UserControl* object and are therefore available to the control only. They neither appear in the finished control's Property window list nor are they available to applications that use the control.

Since the container provides ambient properties, some containers might not implement all the ambient properties. For example, if your ActiveX control is used in an application created in Borland's Delphi, it is possible that all ambient properties won't be available to your control. To circumvent this problem, the *AmbientProperties* object implemented by Visual Basic contains all of the standard ambient properties defined by the ActiveX controls specification, regardless of whether they are actually provided by the container. If a container doesn't implement a particular ambient property, the *AmbientProperties* object simply returns a default value. This is advantageous because you don't have to worry about getting error messages when using an ambient property that is not available.

By the same token, some containers might implement additional ambient properties. These properties will not be visible in the Object Browser because they are not in Visual Basic's type library; nonetheless, the *AmbientProperties* object will expose them. You can learn about such properties in the documentation for a container and then access them as properties of the *AmbientProperties* object. Note that since these additional properties are not in the type library, Visual Basic cannot verify their existence at compile time. Therefore, you should use error handling when working with these properties; it is likely that your control will be used in another container where these properties might not be available.

As a consequence of this lack of type library information, any calls to container-specific ambient properties are late-bound. By contrast, any calls to standard ambient properties are early-bound. Recall that early binding requires type library information to be available at compile time so that all information regarding the object's interface is obtained. Late binding uses the *IDispatch* interface, so no interface information is sought at development time, and thus no error checking can be performed. Instead, all the finding and creating of the object is deferred to run time. In Visual Basic, a call to *CreateObject* always signifies late binding, while setting a reference to an interface by means of the References dialog box indicates early binding.

The standard ambient properties are as follows: *BackColor*, *DisplayAsDefault*, *DisplayName*, *Font*, *ForeColor*, *LocaleID*, *MessageReflect*, *Palette*, *RightToLeft*, *ScaleUnits*, *ShowGrabHandles*, *ShowHatching*, *SupportsMnemonics*, *TextAlign*, *UIDead*, and *UserMode*. In Visual Basic, you can safely ignore many of these ambient properties since default functionality in response to these properties is built in. The following are the ambient properties that you should be aware of:

- *DisplayAsDefault* tells you whether your control is the default button for the container. If your control is the default, pressing the Enter key is equivalent to clicking on the control. Controls that are the default (*DisplayAsDefault* = *True*) should draw an extra-heavy border that identifies the default button to the end user.
- *DisplayName* is the name of the particular instance of the control. It should be used when raising errors at design time so that the developer who receives the error can identify the control instance causing the error.
- *ForeColor*, *BackColor*, and *Font* are hints your control can use to make its appearance match that of the container. For example, in the *InitProperties* event, which is fired whenever an instance of your control is first placed in

a container, it is highly recommended to set your control's *ForeColor*, *BackColor*, and *Font* properties to the values supplied by the ambient properties.

- *TextAlign* tells the control how the client site would like the control to arrange the text it displays. If the value of the property is 0, the control should obey "general" alignment principles: text aligned to the left, numbers to the right. A value of 1 means left alignment, 2 means centered alignment, 3 means right alignment, and 4 means justified alignment.
- *LocaleID* tells the control in which locale (language and country) the control should consider itself to be running.
- *Palette* provides the current palette of the container.
- *UserMode* allows an instance of your control to determine whether it is executing at design time (*UserMode* = *False*) or at run time (*UserMode* = *True*).

The AmbientChanged Event

Assuming that you want to follow the recommendations provided by the ambient properties, what should your control do if one of those ambient properties changes at run time? Let's say you set your control's *BackColor* property based on the ambient *BackColor* property so that your control blends with its container. Eventually, during execution of the application, the background color of the container changes. This causes your control to stick out like a chicken without feathers. To avoid this sort of embarrassing situation, you can write code that responds to the *AmbientChanged* event. The *AmbientChanged* event is fired whenever the value of one of the ambient properties changes, allowing you to respond to this situation gracefully. You declare the *AmbientChanged* event procedure as follows:

```
Private Sub UserControl_AmbientChanged(PropertyName As String)
```

The *PropertyName* argument provides your code with the name of the ambient property whose value changed. This will save you from having to reset all the ambient properties values used in your control.

The Extender Object

Another type of property, known as an *extender property* in Visual Basic, is one that an average user would certainly regard as being associated with the control but that is actually implemented by the container. For example, a control's size and position, its order in the tab sequence, and the *Tag* property fall into the category of extender properties. These properties are associated with something called the *extender object*, which is an object implemented by the container, generally by aggregating with the control. When the container gets or sets a property or invokes a method on the control, the extender object gets access first. If the extender object recognizes the property or method as belonging to it, it performs the actions required; if the extender object doesn't recognize the property or method, it is passed on to the control itself.

A common example of this is the *Enabled* property. The typical visual control will implement this property to reflect the current state of the control. A control, however, knows only its internal state. The control might think it is enabled, but if the control is part of a form that is currently disabled, the control is also disabled.

Note If you are familiar with creating ActiveX controls using Visual C++, you are already familiar with extender properties; in Visual C++ they are called *extended* properties.

The extender object provided by the Visual Basic run time system provides a rich set of properties, methods, and events. The properties it provides are *Cancel*, *Container*, *Default*, *DragIcon*, *DragMode*, *Enabled*, *Height*, *HelpContextID*, *Index*, *Left*, *Name*, *Object*, *Parent*, *TabIndex*, *TabStop*, *Tag*, *ToolTipText*, *Top*, *Visible*, *WhatsThisHelpID*, and *Width*. The methods it provides are *Drag*, *Move*, *SetFocus*, *ShowWhatsThis*, and *ZOrder*. The events it provides are *DragDrop*, *DragOver*, *GotFocus*, and *LostFocus*.

Again, be aware that many containers provide only a limited subset of these properties, methods, and events. The following table shows the extender properties that the ActiveX Controls specification says all containers should provide. All containers should implement these properties, but they aren't required to do so. As a result, when your code refers to extender properties it's a good idea to implement error trapping.

Extender Property	Data Type	Access Attribute	Description
<i>Name</i>	<i>String</i>	Read	The name the container assigns to the control instance.
<i>Visible</i>	<i>Boolean</i>	Read/Write	Indicates whether the control is visible.
<i>Parent</i>	<i>Object</i>	Read	Returns the object that contains the control; for example, a Visual Basic form.
<i>Cancel</i>	<i>Boolean</i>	Read	<i>True</i> if the control is the cancel command button for the container; <i>False</i> if not.
<i>Default</i>	<i>Boolean</i>	Read	<i>True</i> if the control is the default command button for the container; <i>False</i> if not.

Exercise 6-2

Let's create our first test control, which will simply display all the data available in the ambient and extender properties.

1. Open Visual Basic, and start a new Standard EXE project.
2. Choose Add User Control from the Project menu. Select the default User Control, and click Open.
3. In the Properties window for the user control, set the following properties:

Property	Setting
<i>Name</i>	<i>MyControl</i>

<i>AutoRedraw</i>	<i>True</i>
<i>BorderStyle</i>	<i>1 - Fixed Single</i>
<i>Tag</i>	<i>MyControl's Internal Tag</i>

4. Double-click the UserControl designer to open the Code window.
5. In the UserControl_Click event procedure, write the following code (shown in bold):

```
Private Sub UserControl_Click()

    Print "Ambient Properties"
    Print , "LocaleID: " & UserControl.Ambient.LocaleID
    Print , "MessageReflect: " & _
    UserControl.Ambient.MessageReflect
    Print , "TextAlign: " & UserControl.Ambient.TextAlign
    Print , "SupportsMnemonics: " & _
    UserControl.Ambient.SupportsMnemonics
    Print , "BackColor: " & UserControl.Ambient.BackColor
    Print , "DisplayName: " & _
    UserControl.Ambient.DisplayName
    Print , "Font: " & UserControl.Ambient.Font
    Print , "ForeColor: " & UserControl.Ambient.ForeColor
    Print , "ScaleUnits: " & UserControl.Ambient.ScaleUnits
    Print , "TextAlign: " & UserControl.Ambient.TextAlign
    Print , "UserMode: " & UserControl.Ambient.UserMode
    Print , "UIDead: " & UserControl.Ambient.UIDead
    Print , "ShowGrabHandles: " & _
    UserControl.Ambient.ShowGrabHandles
    Print , "ShowHatching: " & _
    UserControl.Ambient.ShowHatching
    Print , "DisplayAsDefault: " & _
    UserControl.Ambient.DisplayAsDefault
    Print , "Palette: " & UserControl.Ambient.Palette

    Print "Extender Properties"
    Print , "Name: " & UserControl.Extender.Name
    Print , "DragIcon: " & UserControl.Extender.DragIcon
    Print , "DragMode: " & UserControl.Extender.DragMode
    Print , "Height: " & UserControl.Extender.Height

    Print , "HelpContextID: " & _
    UserControl.Extender.HelpContextID
    Print , "Left: " & UserControl.Extender.Left
    Print , "TabIndex: " & UserControl.Extender.TabIndex
    Print , "TabStop: " & UserControl.Extender.TabStop
    Print , "Tag: " & UserControl.Extender.Tag
    Print , "Top: " & UserControl.Extender.Top
    Print , "Visible: " & UserControl.Extender.Visible
    Print , "WhatsThisHelpID: " & _
    UserControl.Extender.WhatsThisHelpID
    Print , "Width: " & UserControl.Extender.Width
    Print
    Print "UserControl's Tag:" & UserControl.Tag

End Sub
```

6. Close the UserControl designer. Find the Form window for Form1.
7. Now double-click the MyControl icon in the lower right corner of the toolbox to add a control to the form.
8. Enlarge the form, and then resize the control so that it fills the form.
9. In the Properties window, set the *Tag* property of the form to *My Form's MyControl Tag*.
10. Choose Save Project from the File menu to save your work.
11. Run the program, and click in the control. The output should look similar to Figure 6-2.
12. Observe which *Tag* property is displayed with which object. *My Form's MyControl Tag* is displayed as part of the extender object. *MyControl's Internal Tag* is displayed as part of the *UserControl* object itself.

Adding Custom Properties

Custom properties are by far the most interesting kind of property to talk about, primarily because it is you who implement them. You can name custom properties anything you want, you can accept any parameters you want, and custom properties will generally do your bidding. Controls rarely become useful until you add some custom properties and methods. The simplest type of property to implement is created from a public variable with a line of code such as this:

```
Public PropName As String
```

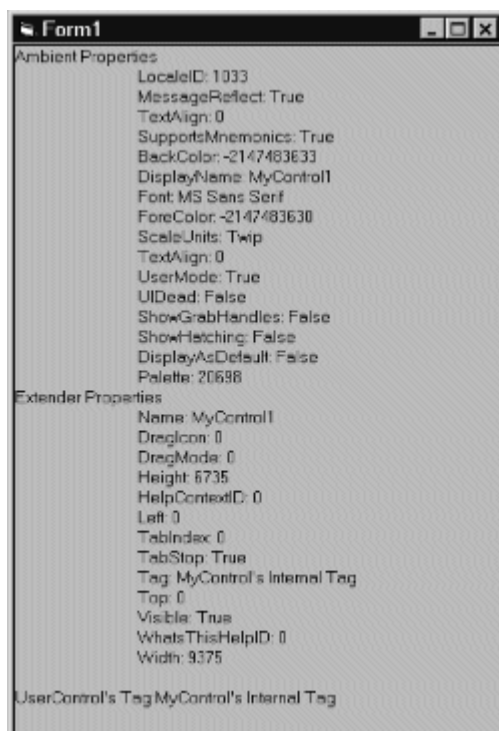


Figure 6-2. The output from MyControl.

Property Procedures

Simple properties declared with the statement *Public PropName As Type* are not a good idea for controls. You should always implement control properties with property procedures instead of with public data members; otherwise, your controls might not work correctly in Visual Basic. You might remember the *Let*, *Get*, and *Set* property procedures that we used in a class module in Chapter 3. Property procedures are required because you must notify Visual Basic whenever a property value changes. You do this by invoking the *PropertyChanged* method of the *UserControl* object whenever a property value has been modified, as is shown in the following code fragment:

```
Private m_MyProperty As String

Public Property Get MyProperty() As String
    MyProperty = m_MyProperty
End Property

Public Property Let MyProperty(ByVal NewValue As String)
    m_MyProperty = NewValue
    UserControl.PropertyChanged "MyProperty"
End Property
```

Without calling the *PropertyChanged* method, Visual Basic does not know that the property has been changed and needs to be saved. Since property values can be displayed in more than one place, the development environment must be notified when a property value changes so that it can synchronize the values. For example, if you have the same property in both a property page and the Properties window, both need to have the same value.

Exercise 6-3

Let's create an ActiveX control with a custom property and property procedures to test this feature.

1. Open Visual Basic, and start a new Standard EXE project.
2. Choose Add User Control from the Project menu. Select the default User Control, and click Open.
3. In the Properties window for the user control, set the properties shown in the following table.

Property	Setting
<i>AutoRedraw</i>	<i>True</i>
<i>BorderStyle</i>	<i>1 - Fixed Single</i>
<i>Name</i>	<i>CustomControl</i>

4. Double-click the UserControl designer to open the Code window.
5. In the Declarations section, add the following code:

```
Option Explicit
Public MyName As String
Private InternalSecondProperty As String

Public Property Get MySecondProperty() As String
```

```

MySecondProperty = InternalSecondProperty
End Property

Public Property Let MySecondProperty(NewValue As String)
    InternalSecondProperty = NewValue
    UserControl.PropertyChanged "MySecondProperty"
End Property

```

6. In the UserControl_Click event procedure, write the following code (shown in bold):

```

Private Sub UserControl_Click()
Print "MyName is " & MyName
Print "MySecondProperty Procedure = " & _
InternalSecondProperty
End Sub

```

7. Close the Code window and the UserControl designer.
8. Find the Form window for Form1. Double-click the CustomControl icon in the toolbox to add the control to the form.
9. Resize the control so that it fills the form.
10. Open the Properties window for CustomControl1. Notice that there is now a MyName property in the property list for CustomControl.
11. Set the MyName property to Hello.
12. Choose Save Project from the file menu, and accept the default names.
13. Run the program, and click the control.
14. Observe that only *MyName is* prints in the control, while *Hello* is missing. Act surprised.
15. Choose End from the Run menu.
16. Return to the Properties window for the *CustomControl1* object, and note that the MyName property is no longer set to Hello. This is because we have not yet implemented property persistence, and property values are not saved. This will be corrected in Exercise 6-4, later in this chapter.
17. Open the Code window for the form Form1, and add the following code (shown in bold) to the Form_Load event procedure:

```

Private Sub Form_Load()
CustomControl1.MyName = "HELLO"
CustomControl1.MySecondProperty = "Hello there"
MsgBox CustomControl1.MySecondProperty
End Sub

```

18. Run the program, click the control, and verify that everything works as expected.
19. When you finish experimenting, choose End from the Run menu.
20. It is enlightening to step through this code line by line using the F8 key. Try this now. Then stop the program.
21. Open the Properties window for the CustomControl1 object. You should see the MySecondProperty property.
22. Save your work.

Property Persistence

Property persistence means keeping property values around for longer than the duration of the program that manipulated them. If you create a program that uses an ActiveX control and you set up the properties of that control to be just right for your application, it'll be a little disconcerting when you start the program the next day and find that all the property values you entered have evaporated. In fact, as you saw in the previous exercise, even setting properties by using the Properties window doesn't work. This will be corrected when you make your properties persistent. Here's how.

ActiveX control modules define three events that relate to property persistence: *InitProperties*, *ReadProperties*, and *WriteProperties*. The *InitProperties* event procedure is called when an ActiveX control is first embedded in a form. This gives you a chance to store default values in your custom properties. The *ReadProperties* event is fired whenever Visual Basic needs to retrieve the value of a custom property—when the project is opened and run, for example. When the *UserControl* object receives its *ReadProperties* event, *siting* has occurred. *Siting* is simply the process of loading a control and placing it in the container. The *WriteProperties* event is fired when a custom property needs to be saved. For example, saving your project always causes the *WriteProperties* event procedure to be called. Here are the declarations for the three property event procedures:

```
Private Sub UserControl_InitProperties()  
Private Sub UserControl_ReadProperties(PropBag As _  
PropertyBag)  
Private Sub UserControl_WriteProperties(PropBag As _  
PropertyBag)
```

In these event procedures, Visual Basic programs use the *PropertyBag* object to implement property persistence. A *PropertyBag* object is just what its name implies—a "bag" in which property values are saved. You can't see into it, and you have no idea where or how the data is saved. (We'll find out shortly.) All you can do is put values in and take them out. The *PropertyBag* class defines two methods, *ReadProperty* and *WriteProperty*, and one property, *Contents*, as follows:

```
ReadProperty(Name As String[, DefaultValue])  
WriteProperty(Name As String, Value[, DefaultValue])  
Property Contents As Variant ' Byte array of PropertyBag  
contents.
```

As you can see, the first argument of the *WriteProperty* method is the name of the property. The *Name* parameter references the property name under which the *Value* argument will be saved. A property value is saved as a *Variant*. Oddly, the third argument is a default value. Why provide a default value when saving a property? Before saving the value, *WriteProperty* compares the default value with the value passed to it for saving.

If the *Value* and *DefaultValue* parameters are identical, nothing is saved. The property value doesn't need to be saved because default values will be set automatically when the control is reloaded. If you consider the great number of properties most controls have, it becomes obvious that storing and then reloading the default values for every property is wasteful. It is usually best to define a global constant, such as *PROPDEFAULT_MYPROPERTYNAME*, to contain the default

value for each property. This is because you need to supply it in three different places: the *InitProperties*, *ReadProperties*, and *WriteProperties* event procedures.

The *ReadProperty* method works similarly. *ReadProperty* accepts the name of the property whose value you want to read and returns that value. The optional *DefaultValue* parameter allows you to specify the default value to be returned in case the property you are looking for has not been previously saved. This will always happen the first time the developer using your control puts your control on a form, since the developer will not have set any properties up to that point. It will also happen if the value has never been changed from the default and therefore has never been saved by *WriteProperty*. Consequently, it is a good idea to include error-trapping code in the *ReadProperties* event procedure; this will protect your control from invalid property values that might have been entered directly into the .FRM file by overzealous users armed with text editors.

Exercise 6-4

1. Start Visual Basic, and open the project you created in Exercise 6-3.
2. Open the CustomControl control in the UserControl designer.
3. Choose Code from the View menu.
4. In the code window for *UserControl*, select the *InitProperties* event procedure and enter the following code (shown in bold):

```
Private Sub UserControl_InitProperties()  
MsgBox "InitProperties"  
MyName = "Default"  
End Sub
```

5. Select the *UserControl_ReadProperties* event procedure, and enter the following code (shown in bold):

```
Private Sub UserControl_ReadProperties(PropBag As _  
PropertyBag)  
MsgBox "ReadProperties"  
MyName = PropBag.ReadProperty("MyName", "Default")  
End Sub
```

6. Select the *UserControl_WriteProperties* event procedure, and enter the following code (shown in bold):

```
Private Sub UserControl_WriteProperties(PropBag As _  
PropertyBag)  
MsgBox "WriteProperties"  
PropBag.WriteProperty "MyName", MyName, "Default"  
End Sub
```

7. Close the Code window and the UserControl designer for CustomControl. If you get a message box, just choose OK to dismiss it.
8. Open Form1 in the Form window.
9. Open the Code window for the form, and comment out the code in the *Form_Load* event procedure:

```
Private Sub Form_Load()  
` COMMENT EVERYTHING OUT  
` CustomControl1.MyName = "HELLO"  
` CustomControl1.MySecondProperty = "Hello there"
```

```

` MsgBox CustomControl1.MySecondProperty
End Sub

```

10. Close the Code window, and open the Properties window for the CustomControl1 object. You should see the MyName property with a *Default* setting caused by the preceding code.
11. Set the MyName property to *Testing*.
12. Run the program. You will get a few message boxes notifying you when properties are being read and written, but everything should work properly.
13. When you are finished experimenting, remove the *MsgBox* statements from the control event procedures and save your work.

What we find really fascinating (and we hope you do too) is that Visual Basic actually executes code in your control even when the test project is not running. This is because even while the test project is in design mode, your control is active and being controlled by Visual Basic. So when Visual Basic needs to read or write property values, it calls the control's event procedures regardless of the current mode. A final thing to look at is the FORM1.FRM file. Open this file in Notepad and you should be able to find the declaration of your CustomControl object and its MyName property value, as shown here:

FORM1.FRM

```

Version 5.00
Begin VB.Form Form1
Caption = "Form1"
... other properties here ...
Begin Project1.CustomControl CustomControl1
Height = 2532
Left = 480
TabIndex = 0
Top = 360
Width = 3732
_ExtentX = 6583
_ExtentY = 4466
MyName = "Testing"
End
End
Attribute VB_Name = "Form1"
Attribute VB_GlobalNameSpace = False
Attribute VB_Creatable = False
Attribute VB_PredeclaredId = True
Attribute VB_Exposed = False
... procedures begin here ...

```

Methods

Methods are procedures built into a control that operate on the control. You can implement methods for ActiveX controls by adding *Public Sub* and *Function* procedures to the *Declarations* section of your control. Although you can create any type of method to perform any task, it is also a good idea to implement methods that are commonly found on controls that provide similar functionality to what you want to do. Since other developers will be familiar with standard methods, they will have an easier time learning to work with your control. For example, it would be poor design to add the functionality of a *Print* method to a

procedure named *Output*. Since Visual Basic programmers are familiar with the *Print* method, they would be more likely to search for *Print* than *Output*.

Recall that Visual Basic's extender object automatically provides five standard methods for every control: *Drag*, *Move*, *SetFocus*, *ShowWhatsThis*, and *ZOrder*. For a control with a visible interface at run time, it is recommended that you also provide a *Refresh* method. Internally, this method should simply call *UserControl.Refresh*. For developer-drawn controls, this will raise the *Paint* event; for controls built using constituent components, it will force a refresh of the constituent controls. In addition, you might want to selectively implement methods of the *UserControl* object or of its constituent controls.

Raising Events from Controls

You need to distinguish between the events received by your ActiveX control and the events your control raises. Events received by your control are opportunities for you to do something interesting; events your control raises provide opportunities for the developer using your control to do something interesting. You might find it helpful to think of properties and methods as inbound and events as outbound. That is, methods are invoked by developers using your control, while events originate in your control and are propagated outward to the developer. This gives the developer using your control a chance to execute code in response to something that happens inside your control.

Two basic steps are required to raise an event from your control. First, you must declare the event itself, using a statement such as this:

```
Event MyEventName()
```

Second, you use the *RaiseEvent* statement to fire the event at the desired time, using a statement such as this:

```
RaiseEvent MyEventName
```

The *RaiseEvent* statement provides functionality similar to simply calling a procedure. However, when calling a procedure, it is required that the called procedure actually exist. Event procedures, on the other hand, cause no errors if they are not implemented—the event is simply ignored. It is also possible to raise events that provide parameters to the event procedure being called. This is useful with events such as *MouseMove*, where more information describing the event that occurred—in this case, the current mouse coordinates—is required. To implement this type of event, simply declare the event as follows:

```
Event MyEventName(MyParameter As Integer)
```

Then you can raise the event with this code:

```
RaiseEvent MyEventName(100)
```

For the benefit of developers using your control, the control's container can provide additional events beyond those you define. As mentioned earlier, Visual Basic's *Extender* object automatically raises four events on behalf of ActiveX controls: *DragDrop*, *DragOver*, *GotFocus*, and *LostFocus*. It is recommended that most controls raise *Click*, *DbClick*, *KeyDown*, *KeyPress*, *KeyUp*, *MouseDown*,

MouseMove, and *MouseUp* events. You'll find it beneficial to implement events commonly found on controls that provide functionality similar to what you want.

Exercise 6-5

Let's create an ActiveX control with methods and events.

1. Open Visual Basic, and start a new Standard EXE project.
2. Choose Add User Control from the Project menu. Select the default User Control, and click Open.
3. In the Properties window for UserControl1, set the properties shown in the following table.

Property	Setting
<i>AutoRedraw</i>	<i>True</i>
<i>BorderStyle</i>	<i>1 - Fixed Single</i>
<i>DrawWidth</i>	<i>2</i>
<i>Name</i>	<i>Scribble</i>

4. Double-click the user control in the UserControl designer to open the Code window.
5. In the Declarations section, add the following code:

```
Option Explicit
Public DrawColor As Long
Event OutOfBounds(X As Single, Y As Single)
```

6. Add the following event procedure code (shown in bold):

```
Private Sub UserControl_MouseDown _
(Button As Integer, Shift As Integer, _
X As Single, Y As Single)
CurrentX = X
CurrentY = Y
End Sub

Private Sub UserControl_MouseMove _
(Button As Integer, Shift As Integer, _
X As Single, Y As Single)
If Button = 1 Then
UserControl.Line _
(CurrentX, CurrentY)-(X, Y), DrawColor

If (X <= 0 Or X >= ScaleWidth Or Y <= 0 Or _
Y >= ScaleHeight) Then
RaiseEvent OutOfBounds(X, Y)
End If
End If
End Sub
```

Exercise 6-5

Let's create an ActiveX control with methods and events.

1. Open Visual Basic, and start a new Standard EXE project.
2. Choose Add User Control from the Project menu. Select the default User Control, and click Open.
3. In the Properties window for UserControl1, set the properties shown in the following table.

Property	Setting
<i>AutoRedraw</i>	<i>True</i>
<i>BorderStyle</i>	<i>1 - Fixed Single</i>
<i>DrawWidth</i>	<i>2</i>
<i>Name</i>	<i>Scribble</i>

4. Double-click the user control in the UserControl designer to open the Code window.
5. In the Declarations section, add the following code:

```
Option Explicit
Public DrawColor As Long
Event OutOfBounds(X As Single, Y As Single)
```

6. Add the following event procedure code (shown in bold):

```
Private Sub UserControl_MouseDown _
(Button As Integer, Shift As Integer, _
X As Single, Y As Single)
CurrentX = X
CurrentY = Y
End Sub

Private Sub UserControl_MouseMove _
(Button As Integer, Shift As Integer, _
X As Single, Y As Single)
If Button = 1 Then
UserControl.Line _
(CurrentX, CurrentY)-(X, Y), DrawColor

If (X <= 0 Or X >= ScaleWidth Or Y <= 0 Or _
Y >= ScaleHeight) Then
RaiseEvent OutOfBounds(X, Y)
End If
End If
End Sub
```

7. Close the Code window and the UserControl designer for the control.
8. Find the Form window for Form1.
9. Add the Scribble control to the form.

10. Add three *OptionButton* controls named Option1, Option2, and Option3 to the form. Set the *Caption* property of the first to *Red*, of the second to *Green*, and of the third to *Blue*.
11. Add the following code (shown in bold) to the *OptionButton* controls' corresponding *Click* event procedures:

```
Private Sub Option1_Click()  
Scribble1.DrawColor = RGB(255, 0, 0)  
End Sub
```

```
Private Sub Option2_Click()  
Scribble1.DrawColor = RGB(0, 255, 0)  
End Sub
```

```
Private Sub Option3_Click()  
Scribble1.DrawColor = RGB(0, 0, 255)  
End Sub
```

12. Place a *CommandButton* control named Command1 on the form, and set its *Caption* property to *Clear Control*.
13. Add the following code (shown in bold) to the command button's *Click* event procedure:

```
Private Sub Command1_Click()  
Scribble1.Cls  
End Sub
```

14. Place a *TextBox* control named Text1 on the form, and clear its *Text* property.
15. Double-click the Scribble control to open the Code window. Add the following code to the Scribble1_OutOfBounds event procedure (shown in bold):

```
Private Sub Scribble1_OutOfBounds(X As Single, Y As Single)  
Text1 = "OutOfBounds " & X & ", " & Y  
End Sub
```

16. Choose Save Project from the File menu. Accept the default filenames.
17. Choose Start from the Run menu, and test the program. Notice that the *OutOfBounds* coordinates display in the text box when you drag the mouse from a point within the Scribble control past one of its edges.

Handling Errors in Controls

You manage errors in ActiveX controls the same as you do for ActiveX code components, but we have one warning: you should never raise errors within an event procedure since there is nowhere for these errors to be trapped. For example, suppose a *Click* event procedure is fired when the user clicks a control. If the control raises an error in response to this event, the container of the form never has a chance to respond. Visual Basic will simply report the errors and stop the application.