This chapter covers control creation in depth. The majority of the topics are organized according to the sequence of control development tasks outlined in “Control Creation Recap,” at the end of the topic, “Creating an ActiveX Control.”

First and most important, however, is an introduction to the terminology and concepts of control creation, in the topics “Control Creation Terminology,” “Control Creation Basics,” and “Interacting with the Container.”

These are followed by topics associated with development tasks:

1. Determine the features your control will provide.

   1“Visual Basic ActiveX Control Features.”

2. Design the appearance of your control.

   2“Drawing Your Control.”

3. Design the interface for your control — that is, the properties, methods, and events your control will expose.

   3“Adding Properties to Controls,” “Adding Methods to Controls,” “Raising Events from Controls,” and “Providing Named Constants for Your Control.”

4. Create a project group consisting of your control project and a test project.

   4“Setting Up a New Control Project and Test Project.”

5. Implement the appearance of your control by adding controls and/or code to the UserControl object.

6. Implement the interface and features of your control.

   5“Creating Robust Controls.”

7. As you add each interface element or feature, add features to your test project to exercise the new functionality.

   6“Debugging Controls.”

8. Design and implement property pages for your control.

   7This subject is covered in “Creating Property Pages for ActiveX Controls.”

9. Compile your control component (.ocx file) and test it with all potential target applications.

   8“Distributing Controls.”

1

The chapter ends with “Localizing Controls,” which discusses localizing your control for other languages. The complete list of top-level topics is:

Contents

- Visual Basic ActiveX Control Features
- Drawing Your Control
- Adding Properties to Controls
- Adding Methods to Controls
- Raising Events from Controls
- Providing Named Constants for Your Control
- Setting Up a New Control Project and Test Project
- Creating Robust Controls
- Debugging Controls
- Distributing Controls
- Localizing Controls

1

Sample Application: CtlPlus.vbg

Includes a fully functional version of the ShapeLabel control created in the step by step procedures in Chapter 4, “Creating an ActiveX Control,” and other controls that illustrate the control creation features in this chapter. If you installed the sample applications, you will find CtlPlus.vbg in the \CompTool\ActvComp subdirectory of the Visual Basic samples directory (\Vb\Samples\CompTool\ActvComp).

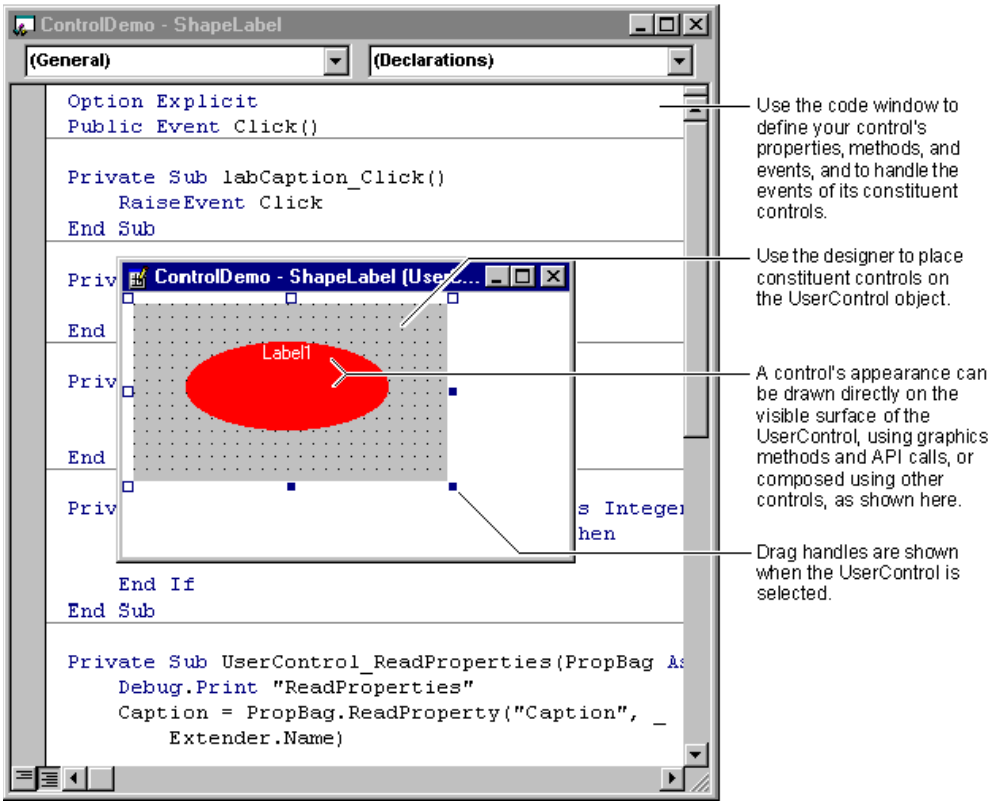
The UserControl Object

An ActiveX control created with Visual Basic is always composed of a *UserControl object,* plus any controls — referred to as *constituent controls* — that you choose to place on the UserControl.

Like Visual Basic forms, UserControl objects have code modules and visual designers, as shown in Figure 9.1. You place constituent controls on the UserControl object’s designer, just as you would place controls on a form.

—2

Figure 9.1 UserControl designer and code window

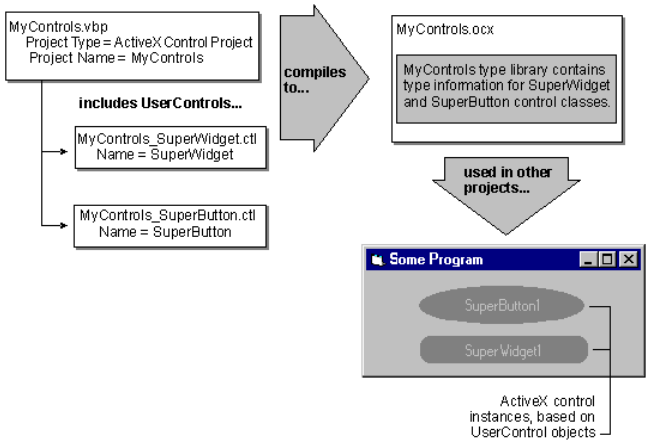


2

Like forms, UserControls are stored in plain text files that contain the source code and property values of the UserControl and its constituent controls. Visual Basic uses the extension .ctl for these source files.

The relationship of .ctl files and ActiveX control projects to finished controls and .ocx files is shown in Figure 9.2.

Figure 9.2 ActiveX control projects are built into .ocx files



3

If a UserControl or its constituent controls use graphical elements which cannot be stored as plain text, such as bitmaps, Visual Basic stores those elements in a .ctx file with the same name you give to the .ctl file. This is analogous to the .frx files used to store graphical elements used in forms.

The .ctl and .ctx files completely define an ActiveX control's appearance and interface (properties, methods, and events). You can include .ctl files in any of the project types. "Two Ways to Package ActiveX Controls," later in this chapter, discusses this subject in depth.

Delegating to the UserControl and Constituent Controls that Compose Your ActiveX Control

Your ActiveX control is said to be *composed* of a UserControl and its constituent controls because each instance will actually contain those objects.

That is, whenever you place an instance of your ActiveX control on a form, a UserControl object is created, along with instances of any constituent controls you placed on the UserControl designer. These objects are *encapsulated* inside your control.

The UserControl object has an interface — that is, properties, methods, and events — of its own. The interface of your ActiveX control can *delegate* to the UserControl object's interface members, which are hidden from the user of your control by encapsulation.

That is, rather than writing your own code to implement a BackColor property, you can delegate to the UserControl object's BackColor property, and let it do all the work. In practice, this means that the BackColor property of your ActiveX control simply calls the BackColor property of the UserControl object.

In the same manner, you can piggy-back your control's Click event on the existing functionality of the UserControl object's Click event.

The interface for your ActiveX control can also delegate to the properties, methods, and events of the constituent controls you place on the UserControl designer, as discussed in "Exposing Properties of Constituent Controls," "Adding Methods to Controls," and "Exposing Events of Constituent Controls," later in this chapter.

For More Information For a discussion of what controls you can place on a UserControl designer, see "Controls You Can Use As Constituent Controls," later in this chapter.

2

Three Ways to Build ActiveX Controls

There are three models for control creation in Visual Basic. You can:

- Author your own control from scratch.
- Enhance a single existing control.
- Assemble a new control from several existing controls.

4

The second and third models are similar, because in both cases you put constituent controls on a UserControl object. However, each of these models has its own special requirements.

Authoring a User-Drawn Control

Writing a control from scratch allows you to do anything you want with your control's appearance and interface. You simply put code into the Paint event to draw your control. If your control's appearance changes when it's clicked, your code does the drawing.

This is the model you should select if you're creating a new visual widget, such as a button that crumbles to dust and disappears when clicked.

For More Information Creating a user-drawn control is discussed further in "Drawing Your Control," later in this chapter.

3

Enhancing an Existing Control

Enhancing an existing control means putting an instance of the control on a UserControl designer and adding your own properties, methods, and events.

You have complete freedom in specifying the interface for your enhanced control. The properties, methods, and events of the control you start with will only be included in your interface if you decide to expose them.

"Exposing Properties of Constituent Controls," later in this chapter, describes how to do this manually, and how to make it easier by using the ActiveX Control Interface Wizard.

—5

Enhancing the appearance of an existing control is more difficult than enhancing the interface, because the control you're enhancing already contains code to paint itself, and its paint behavior may depend on Windows messages or other events.

Experienced Windows programmers can subclass the constituent control using the AddressOf operator described in "Using the Windows API," in the *Component Tools Guide*. This allows some control over the control's appearance, but there is no way to alter the control's paint code.

It's easier to work with the control's built-in paint behavior, and instead enhance it by adding properties, methods, and events, or by intercepting and altering existing properties and methods. This is discussed further in "Drawing Your Control," later in this chapter.

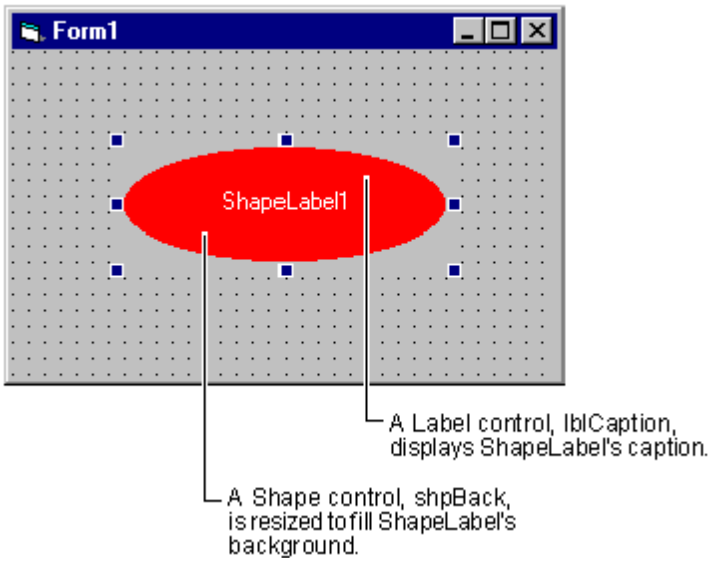
Assembling a Control from Several Existing Controls

You can construct your control's appearance and interface quickly by assembling existing controls on a UserControl designer.

For example, the ShapeLabel control provided in the CtlPlus.vbg sample application, and discussed in the step by step procedures in "Creating an ActiveX Control," uses a Shape control to provide its visual background and a Label control to display its caption.

Figures 9.3 and 9.4 show how multiple constituent controls can contribute to the appearance and interface of an ActiveX control.

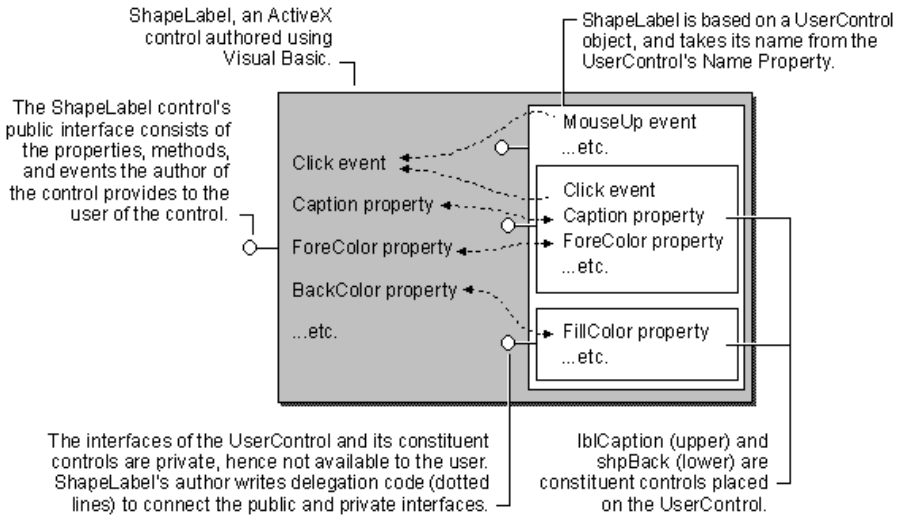
Figure 9.3 Constituent controls provide ShapeLabel's appearance



5

Constituent controls contribute to the appearance of an instance of your control by their mere presence on the UserControl designer. They contribute to your control's interface by *delegation,* as shown in Figure 9.4.

Figure 9.4 Constituent controls contribute to ShapeLabel's interface



6

For example, ShapeLabel's Caption property delegates to the Caption property of the constituent control lblCaption as shown in the following code fragment.

```
Public Property Get Caption() As String
    Caption = lblCaption.Caption
End Property

Public Property Let Caption(NewCaption As String)
    lblCaption.Caption = NewCaption
    PropertyChanged "Caption"
End Property
```

4

For More Information "Drawing Your Control" and "Exposing Properties of Constituent Controls," later in this chapter, discuss control assemblies in more depth. The purpose and importance of PropertyChanged are discussed in "Adding Properties to Controls," later in this chapter.

5

Two Ways to Package ActiveX Controls

An ActiveX control created with Visual Basic is defined by a UserControl module. The source code you add to this module, to implement your ActiveX control, is stored in a .ctl file.

You can include UserControl modules in most Visual Basic project types, but only ActiveX control projects can provide controls to other applications. Controls in all other project types are private.

Thus there are two ways to package controls:

- *Public controls* can only exist in ActiveX control projects. You make a control public by setting the Public property of the UserControl object to True.

  9Public controls can be used by other applications, once the ActiveX control project has been compiled into a control component (.ocx file).

- *Private controls* can exist in any project type. You make a control private by setting the Public property of the UserControl object to False.

  10After the project is compiled, private controls cannot be used by other applications. They can be used only within the project in which they were compiled.

  11If you attempt to set the Public property of a UserControl object to True, and the UserControl is not in an ActiveX control project, an error occurs.

7

If one of the controls in an ActiveX control project is meant to be used only as a constituent of other controls in the project, you can set the Public property of the UserControl to False. The control will then be available only to the controls of which it is a constituent part. Other applications will not be able to use it.

Note You cannot include UserControl modules in a project marked for unattended execution. If the Unattended Execution box is checked on the General tab of the Project Properties dialog box, the project cannot contain any user interface elements.

6

Including Controls as Compiled Code vs. Source Code

If you create your controls as public classes in an ActiveX control project, you can distribute the compiled control component (.ocx file) with any application you create. When you use SetupWizard to create a setup program for an application in which you've used such a control, the compiled .ocx file will be included automatically.

You can also create a setup program for the control component itself, and distribute it to other developers. "Licensing Issues for Controls," later in this chapter, discusses the licensing support available for control components authored using Visual Basic.

Changing the Packaging

Once you've authored a control, you can easily change the way the control is packaged.

For example, if you have some private controls that are part of a Standard EXE project, and you want to allow other applications to use them, you can add the .ctl

files to an ActiveX control project, and compile it into a distributable control component (.ocx file).

Source Code

Instead of including the compiled control component in your applications, you can simply add the .ctl file to the project for the application. When the application is compiled, the control is compiled into the executable.

The primary advantages of including a control as source code are:

- There is no additional .ocx file to distribute.
- You don't have to debug your control for all possible test cases. You only have to debug the features used by your application.
- You don't have to worry about whether your application will work with future versions of the control, because the version your application uses is compiled in.

8

Note Some developers may argue that avoiding the additional .ocx file is not really an advantage. All Visual Basic applications require support files, and SetupWizard automatically includes them in your setup program, so you're not avoiding any extra work.

7

Of course, there's no such thing as a free lunch. There are also disadvantages to including controls as source code:

- If you discover a bug in the control, you cannot simply distribute an updated .ocx file. You must recompile the entire application.
- Multiple applications will require more disk space, because instead of sharing one copy of an .ocx file, each application includes all the code for the control.
- Each time you use the source code in an application, there will be an opportunity to fix bugs or enhance the code. It may become difficult to keep track of which version of a control was used in which version of which application.
- Sharing source code with other developers may be problematic. At the very least, it's likely to require more support effort than distributing a compiled component. In addition, you give up control and confidentiality of your source code.

9

Understanding Control Lifetime and Key Events

Designing ActiveX controls involves a radical shift in perspective. The key events you must respond to are different — for example, your life will revolve around the Resize event — and there's no such thing as QueryUnload. But that's just the beginning.

"Control Creation Terminology," earlier in this chapter, introduced the idea that a control is not a permanent fixture of a form. Indeed, design-time and run-time

instances of your control will be created and destroyed constantly — when forms are opened and closed, and when you run the project.

Each time an instance of your ActiveX control is created or destroyed, the UserControl object it's based on is created or destroyed, along with all of its constituent controls. ("The UserControl Object," earlier in this chapter, explains the basis of all ActiveX controls created with Visual Basic.)

Consider, for example, a day in the life of the ShapeLabel control used in the step by step procedures in "Creating an ActiveX Control."

10. The user creates an instance of ShapeLabel — by double-clicking on the Toolbox, or by opening a form on which an instance of ShapeLabel was previously placed.
11. The constituent controls, a Shape and a Label, are created.
12. The UserControl object is created, and the Shape and Label controls are sited on it.
13. The UserControl_Initialize event procedure executes.
14. The ShapeLabel control is sited on the form.
15. If the user is placing a new ShapeLabel, the InitProperties event of the UserControl object occurs, and the control's default property values are set. If an existing form is being opened, the ReadProperties event occurs instead, and the control retrieves its saved property values.
16. The UserControl_Resize event procedure executes, and the constituent controls are resized according to the size the user made the new control instance, or the size they were before the form was closed.
17. The Show and Paint events occur. If there are no constituent controls, the UserControl object draws itself.
18. The user presses F5 to run the project. Visual Basic closes the form.
19. The UserControl object's WriteProperties event occurs, and the control's property values are saved to the in-memory copy of the .frm file.
20. The control is unsited.
21. The UserControl object's Terminate event occurs.
22. The UserControl object and its constituent controls are destroyed.

10

And that's not the half of it. The run-time instance of the form is now created, along with a run-time instance of the ShapeLabel control. When the user closes the form and returns to design mode, the ShapeLabel is destroyed and re-created once again.

The rest of this topic explains the key events in a UserControl object's life, and provides reference lists of the events you receive in several important scenarios.

Key UserControl Events

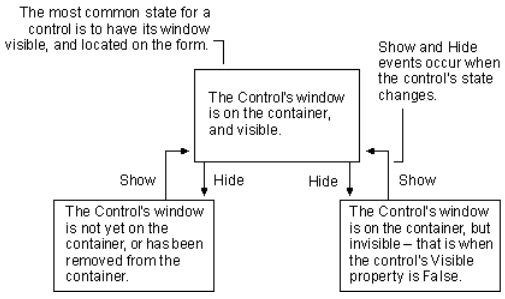
The meanings of the key events in the life of a UserControl object are as follows:

- The Initialize event occurs every time an instance of your control is created or re-created. It is always the first event in a control instance's lifetime.
- The InitProperties event occurs only in a control instance's first incarnation, when an instance of the control is placed on a form. In this event, you set the initial values of the control's properties.
- The ReadProperties event occurs the second time a control instance is created, and on all subsequent re-creations. In this event, you retrieve the control instance's property values from the in-memory copy of the .frm file belonging to the form the control was placed on.
- The Resize event occurs every time a control instance is re-created, and every time it is resized — whether in design mode, by the developer of a form, or at run time, in code. If your UserControl object contains constituent controls, you arrange them in the event procedure for this event, thus providing your control's appearance.
- The Paint event occurs whenever the container tells the control to draw itself. This can occur at any time, even before the control receives its Show event — for example, if a hidden form prints itself. For user-drawn controls, the Paint event is where you draw your control's appearance.
- The WriteProperties event occurs when a *design-time* instance of your control is being destroyed, if at least one property value has changed. In this event, you save all the property values a developer has set for the control instance. The values are written to the in-memory copy of the .frm file.
- The Terminate event occurs when the control is about to be destroyed.

11

In addition to the events listed above, the Show and Hide events may be important to your control. Show and Hide occur as indicated in Figure 9.5.

Figure 9.5 Show and Hide Events



12

In order to draw to the screen in Windows, any control must have a window, temporarily or permanently. Visual Basic ActiveX controls have permanent windows. Before a control has been sited on a form, its window is not on the container. The UserControl object receives Show and Hide events when the window is added and removed.

While the control's window is on the form, the UserControl receives a Hide event when the control's Visible property changes to False, and a Show event when it changes to True.

The UserControl object does *not* receive Hide and Show events if the form is hidden and then shown again, or if the form is minimized and then restored. The control's window remains on the form during these operations, and its Visible property doesn't change.

If the control is being shown in an internet browser, a Hide event occurs when the page is moved to the history list, and a Show event occurs if the user returns to the page.

Note If your control is used with earlier versions of Visual Basic, the UserControl object will not receive Show and Hide events at design time. This is because earlier versions of Visual Basic did not put any visible windows on a form at design time.

8

For More Information The topic "Life and Times of a UserControl Object," one of the step by step procedures in "Creating an ActiveX Control," demonstrates the key events in the life of a control and illustrates how often control instances are created and destroyed.

9

The Incarnation and Reincarnation of a Control Instance

Let's follow a control instance from its placement on a form, through subsequent development sessions, until it's compiled into an application. We'll assume the control was already developed and compiled into an .ocx file, before the curtain opens.

The scenarios that follow mention both Resize and Paint events. Which event you're interested in depends on the control creation model you're using, as discussed in "Three Ways to Build ActiveX Controls," earlier in this chapter.

If your control provides its appearance using constituent controls, you'll use the Resize event to size the constituent controls. If you're authoring a user-drawn control, on the other hand, you can ignore the Resize event and remarks about constituent controls. User-drawn controls draw their appearance in the Paint event. This is discussed in "Drawing Your Control," later in this chapter.

Note In all of these scenarios, the order and number of Resize and Paint events may vary.

10

The Control Instance is Placed on a Form

When you double-click a control's icon in the Toolbox, a design-time instance of the control is placed on the form you're designing. The following events occur in the UserControl object at the heart of the control instance:

—12

Event	What gets done
Initialize	Constituent controls have been created, but the control has not been sited on the form.
InitProperties	The control instance sets default values for its properties. The control has been sited, so the Extender and Ambient objects are available. This is the only time the instance will ever get this event.
Resize, Paint	The control instance adjusts the size of its constituent controls, if any, according to its default property settings. A user-drawn control draws itself.

11

The developer of the form can now see the control, and set its properties in the Properties window. After the developer does this, she may press F5 to run the project.

From Design Mode to Run Mode

When F5 is pressed, the control's design-time instance on the form is destroyed. When the form is loaded at run time, the control is recreated as a run-time instance.

Event	What gets done
WriteProperties	Before the design-time instance is destroyed, it has a chance to save property values to the in-memory copy of the .frm file.
Terminate	Constituent controls still exist, but the design-time control instance is no longer sited on the form. It's about to be destroyed.
Initialize	Constituent controls have been created, but the run-time control instance has not been sited on the form.
ReadProperties	The control instance reads the property values that were saved in the in-memory .frm file. The control has been sited on the run-time instance of the form, so the Extender and Ambient objects are available.
Resize, Paint	The control instance adjusts the size of its constituent controls, if any, according to its current property settings. A user-drawn control draws itself.

12

The developer tests the form by clicking the control, or taking other actions that cause the control's properties, methods, and events to be exercised.

From Run Mode to Design Mode

Finally the developer closes the form and returns to design mode. The run-time instance of the control is destroyed, and a design-time instance is created:

Event	What gets done
Terminate	The run-time instance never gets a chance to save property settings. Changes to property values while the program was running are discarded.
Initialize	Design-time instances of constituent controls have been created, but the design-time control instance has not been sited on the form.
ReadProperties	The control reads the property values that were saved in the in-

	memory copy of the .frm file. The control has been sited on the design-time instance of the form, so the Extender and Ambient objects are available.
Resize, Paint	The control instance adjusts the size of its constituent controls, if any, according to its saved property settings. A user-drawn control draws itself.

13

Closing the Form

If the developer doesn't need to work on the form any more, she may close it. Or it may be quitting time, and she may close the whole project. In either case, the control instance on the form is destroyed.

Event	What gets done
WriteProperties	Before the design-time instance is destroyed, it has a chance to save property values to the in-memory copy of the .frm file.
Terminate	Constituent controls still exist, but the control instance is no longer sited on the form. It's about to be destroyed.

14

Note In all of the scenarios above, the control instance has been saving its property values to the in-memory copy of the .frm file. If the developer chooses not to save the project before closing it, those property settings will be discarded.

15

Additional Scenarios

When the developer re-opens the project, and opens the form to work on it again, the control is reincarnated as a design-time instance. It receives Initialize, ReadProperties, Resize, Paint, and WriteProperties events.

Note A *WriteProperties* event? Yes, indeed. When the project is opened, Visual Basic creates an in-memory copy of the .frm file. As each control on the form is created, it gets a ReadProperties event to obtain its saved property values from the .frm file, and a WriteProperties event to write those property values to the in-memory copy of the .frm file.

16

Compiling the Project

When the project is compiled into an application or component, Visual Basic loads all the form files invisibly, one after another, in order to write the information they contain into the compiled file. A control instance gets the Initialize, ReadProperties, and WriteProperties events. The control's property settings are compiled into the finished executable.

Running the Compiled Program or Component

Whenever a user runs the installed application or component, and the form is loaded, the control receives Initialize, ReadProperties, and Resize events. When the form is unloaded, the control receives a Terminate event.

Controls on World Wide Web Pages

Unlike Visual Basic projects and compiled programs, HTML pages don't save design-time information. Therefore a control on an HTML page always acts as though it's being created for the very first time. When the HTML is processed by a browser, a control on the page receives the Initialize, InitProperties, Resize, and Paint events.

Property values specified with the <param name> tag, between the <OBJECT> and </OBJECT> tags that specify the control's place on the page, are assigned once the control is running, as discussed in "Adding Internet Features to Controls," later in this chapter.

Events You Won't Get in a UserControl object

Some events you're familiar with from working with forms don't exist in a UserControl object. For example, there is no Activate or Deactivate event, because controls are not activated and deactivated the way forms are.

More striking is the absence of the familiar Load, Unload, and QueryUnload events. Load and Unload simply don't fit the UserControl lifestyle; unlike a form, a control instance isn't loaded at some point after it's created — when a UserControl object's Initialize event occurs, constituent controls have already been created.

The UserControl object's Initialize and ReadProperties events provide the functionality of a form's Load event. The main difference between the two is that when the Initialize event occurs, the control has not been sited on its container, so the container's Extender and Ambient objects are not available. The control has been sited when ReadProperties occurs.

Note ReadProperties doesn't occur the first time a control instance is placed on a container — in that case the InitProperties occurs instead.

17

The UserControl event most like a form's Unload event is Terminate. The constituent controls still exist at this point, although you no longer have access to the container, because your control has been unsited.

The WriteProperties event cannot be used as an analog of Unload, because it occurs only at design time.

UserControl objects don't have QueryUnload events because controls are just parts of a form; it's not up to a control to decide whether or not the form that contains it should close. A control's duty is to destroy itself when it's told to.

Events Peculiar to UserControls

The GotFocus and LostFocus events of the UserControl object notify user-drawn controls when they should show or stop showing a focus rectangle. These events should not be forwarded to the user of your control, because the container is responsible for focus events.

If your UserControl has constituent controls that can receive focus, the EnterFocus event will occur when the first constituent control receives the focus, and the ExitFocus event will occur when focus leaves the last constituent control. See “How to Handle Focus in your Control,” later in this chapter.

If you have allowed developers to set access keys for your control, the AccessKeyPress event occurs whenever a user presses an access key. See “Allowing Developers to Set Access Keys for Your Control,” later in this chapter. The AccessKeyPress event can also occur if your control is a default button or cancel button. This is discussed in “Allowing Your Control to be a Default or Cancel Button,” later in this chapter.

The AmbientChanged event occurs whenever an Ambient property changes on the container your control has been placed on. See “Using the Ambient Object to Stay Consistent with the Container,” later in this chapter.

Interacting with the Container

As explained in “Control Creation Terminology,” earlier in this chapter, instances of your control never exist by themselves. They are always placed on container objects, such as Visual Basic forms.

Container objects supply additional properties, methods, and events that appear to the user to be part of your control. This is discussed in the related topic, “Understanding the Container’s Extender Object.” You can use the Parent property of the Extender object to access the properties and methods of the container your control has been placed on.

You can also obtain information about the container through the UserControl object’s Ambient property. The object returned by this property offers hints for property settings, such as BackColor, that can make your control’s appearance consistent with that of its container. The Ambient object is discussed in “Using the Ambient Object to Stay Consistent with the Container.”

Note The Ambient and Extender objects are not available until your control has been sited on the container. Thus they are not available in the UserControl object’s Initialize event. When the InitProperties or ReadProperties event occurs, the control instance has been sited.

18

All Containers are Not Created Equal

A consequence of your control’s dependence on container objects is that some features may not be available in all containers. Many ActiveX control features require support from the container a control is placed on, and will be disabled if the container doesn’t provide the required support.

The following features are supported by Visual Basic forms, but may not be supported by all containers:

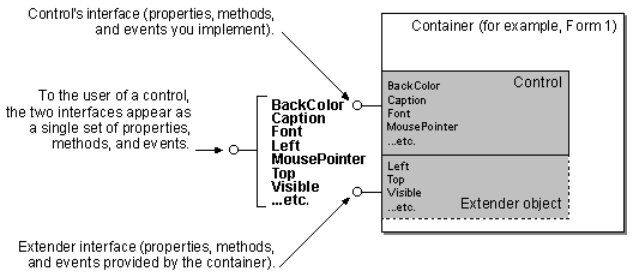
- Transparent control background, discussed in “Giving Your Control a Transparent Background,” later in this chapter.
- The ControlContainer property, discussed in “Allowing Developers to Put Controls on Your Control,” later in this chapter.
- Alignable controls, discussed in “Making Your Control Align to the Edges of Forms,” later in this chapter.
- Modeless dialog boxes your control may show.

13

Understanding the Container’s Extender Object

When you view an instance of your control in the Properties window, you’ll see a number of properties you didn’t author. These *extender properties* are provided by the container your control is placed on, but they appear to be a seamless extension of your control, as shown in Figure 9.6.

Figure 9.6 Extender properties, methods, and events are provided by the container



14

A UserControl object can access extender properties through its Extender object. For example, the ShapeLabel control in “Creating an ActiveX Control” uses the following code to initialize its Caption property:

```
Private Sub UserControl_InitProperties()
    ' Let the starting value for the Caption
    '   property be the default Name of this
    '   instance of ShapeLabel.
    Caption = Extender.Name
End Sub
```

19

Extender properties are provided for the developer who uses your control. Generally speaking, the author of a control should not attempt to set them with code in the UserControl. For example, it’s up to the developer to decide where a particular instance of your control should be located (Top and Left properties), or what icon it should use when dragged.

Extender Properties are Late Bound

When you compile your control component, Visual Basic has no way of knowing what kind of container it may be placed on. Therefore references to Extender properties will always be late bound.

Standard Extender Properties

The ActiveX control specification lists the following properties that all containers should provide:

Property	Type	Access	Meaning
Name	String	R	The name the user assigns to the control instance.
Visible	Boolean	RW	Indicates whether the control is visible.
Parent	Object	R	Returns the object which contains the control, such as a Visual Basic form.
Cancel	Boolean	R	True if the control is the cancel button for the container.
Default	Boolean	R	True if the control is the default button for the container.

20

Although it is highly recommended that containers implement these properties, containers do not have to do so. Thus you should always use error trapping when referring to properties of the Extender object in your code, even standard properties.

Many containers provide additional extender properties, such as Left, Top, Width, and Height properties.

Note If you wish your control to be invisible at run time, set the UserControl object's InvisibleAtRuntime property to True, as discussed in "Making Your Control Invisible at Run Time," later in this chapter. Do not use the Extender object's Visible property for this purpose.

21

Container-Specific Controls

If you design your control so that it requires certain Extender properties, your control will not work in containers that don't provide those properties. There is nothing wrong with building such *container-specific* controls, except that the potential market for them is smaller.

If you are creating a control designed to address a limitation of a particular container, such considerations may not matter to you. However, conscientious use of error trapping will prevent your control from causing unfortunate accidents if it is placed on containers it was not specifically designed for.

Working with Container Limitations

Visual Basic provides a rich set of extender properties and events, listed in Help for the Extender object. Many containers provide only a limited subset of these.

In general, Extender properties, methods, and events are not the concern of the control author. Many Extender properties, such as Top and Left, or WhatsThisHelpID, cannot be implemented by a control, because the container must provide the underpinnings these properties require.

Collisions Between Control and Extender Properties

If an instance of your control is placed on a container that has an extender property with the same name as a property of your control, the user will see the extender property.

For example, suppose you gave your control a Tag property. When an instance of your control is placed on a Visual Basic form, a Tag property is supplied by the form's Extender object. If your control is called ShapeLabel, the user might write the following code:

```
ShapeLabel1.Tag = "Triceratops"
```

22

The code above stores the string "Triceratops" in the Tag property provided by the Visual Basic form's Extender object. If an instance of your control is placed on a container whose Extender object doesn't supply a Tag property, the same code will store the string in the Tag property you implemented.

In order to access the Tag property of your control on a Visual Basic form, the user could employ another Extender object property, as shown in the following code fragment:

```
ShapeLabel1.Object.Tag = "Triceratops"
```

23

The Object property returns a reference to your control's interface just as you defined it, without any extender properties.

Using the Ambient Object to Stay Consistent with the Container

Containers provide *ambient properties* to give controls hints about how they can best display themselves in the container. For example, the ambient BackColor property tells a control what color to set its own BackColor property to in order to blend in with the container.

Visual Basic makes ambient properties available to your ActiveX control through an Ambient object. The Ambient property of the UserControl object returns a reference to the Ambient object.

The Ambient object provided by Visual Basic contains all of the standard ambient properties defined by the ActiveX Controls Standard, whether or not they are actually provided by the container your control instance was placed on.

This means that you can safely access any of the properties of the Ambient object visible in the Object Browser. If you access an ambient property not provided by the

container, the Ambient object returns a default value, as listed in Help for the Ambient object properties.

Containers That Provide Additional Ambient Properties

Control containers may define their own ambient properties. These container-specific ambient properties are not visible in the Object Browser, because they are not in Visual Basic's type library. You can learn about such properties in the documentation for a container, and access them as if they were properties of the Ambient object.

Because these properties are not in the type library, Visual Basic cannot verify their existence at compile time. Therefore you should always use error handling when working with Ambient properties.

Another consequence of the lack of type library information is that calls to container-specific ambient properties are always late-bound. By contrast, calls to standard ambient properties are early-bound.

Important Ambient Properties

You can ignore many of the standard ambient properties. In a Visual Basic ActiveX control, you can ignore the MessageReflect, ScaleUnits, ShowGrabHandles, ShowHatching, SupportsMnemonics, and UIDead properties of the Ambient object.

Ambient properties you should be aware of are listed below.

UserMode

The most important property of the Ambient object is UserMode, which allows an instance of your control to determine whether it's executing at design time (UserMode = False) or at run time. Use of this property is discussed in "Creating Design-Time-Only or Run-Time-Only Properties," later in this chapter.

Tip To remember the meaning of UserMode, recall that at design time the person working with your control is a *developer,* rather than an end user. Thus the control is not in "user" mode, so UserMode = False.

24

LocaleID

If you're developing a control for international consumption, you can use the LocaleID ambient property to determine the locale. Use of this property is discussed in "Localizing Your Control," later in this chapter.

DisplayName

Include the value of the DisplayName property in errors your control raises at design-time, so the developer using your control can identify the control instance that is the source of the error.

ForeColor, BackColor, Font, and TextAlign

These properties are hints your control can use to make its appearance match that of the container. For example, in the InitProperties event, which each instance of your

UserControl receives when it is first placed on a container, you can set your control's ForeColor, BackColor, Font, and TextAlign to the values provided by the ambient properties. This is a highly recommended practice.

You could also give your control properties which the user could use to keep a control instance in sync with the container. For example, you might provide a MatchFormBackColor property; setting this property to True would cause your control's BackColor property always to match the value of the BackColor property of the Ambient object. You can provide this kind of functionality using the AmbientChanged event, discussed below.

DisplayAsDefault

For user-drawn controls, this property tells you whether your control is the default button for the container, so you can supply the extra-heavy border that identifies the default button for the end user.

If you didn't set your control up to be a default button, you can ignore this property. See "Allowing Your Control to be a Default or Cancel Button," in this chapter.

For More Information See "User-Drawn Controls," in this chapter.

25

The AmbientChanged Event

If your control's appearance or behavior is affected by changes to any of the properties of the Ambient object, you can place code to handle the change in the UserControl_AmbientChanged event procedure.

The argument of the AmbientChanged event procedure is a string containing the name of the property that changed.

Important If you're authoring controls for international use, you should always handle the AmbientChanged event for the LocaleID property. See "Localizing Controls," later in this chapter.

26

Note If an instance of your control is placed on a Visual Basic form, and the FontTransparent property of the form is changed, the AmbientChanged event will not be raised.

27

Visual Basic ActiveX Control Features

Visual Basic allows you to author full-featured ActiveX controls. The topics included in this section explain many of the features of the UserControl object that enable ActiveX control capabilities.

You can read about other potential ActiveX control capabilities in "Adding Properties to Controls," "Adding Methods to Controls," and "Raising Events from Controls."

You can also add property pages to your control, as discussed in "Creating Property Pages for ActiveX Controls."

Many of the features described in this section are also demonstrated in the CtlPlus.vbg sample application, which you can find in \VB\Samples\CompTool\ActvComp.

How to Handle Focus in Your Control

The way you handle focus for your control depends on which model you're using to develop your control. Models for building ActiveX controls are discussed in "Three Ways to Build ActiveX Controls," earlier in this chapter.

User-Drawn Controls

If you're authoring a user-drawn control, there won't be any constituent controls on your UserControl. If you don't want your control to be able to receive the focus, set the CanGetFocus property of the UserControl object to False. CanGetFocus is True by default.

If your user-drawn control can receive the focus, it will receive GotFocus and LostFocus events when it receives and loses the focus. A user-drawn control is responsible for drawing its own focus rectangle when it has the focus, as described in "User-Drawn Controls," in this chapter.

This is the only function your UserControl's GotFocus and LostFocus events need to fulfill for a user-drawn control. You don't need to raise GotFocus or LostFocus events for the user of your control, because the container's extender provides these events.

Note The UserControl object of a user-drawn control will also receive a EnterFocus event prior to GotFocus, and an ExitFocus event after LostFocus. You don't need to put any code in the event procedures of these event, and in fact it is recommended that you not do so.

28

Controls That Use Constituent Controls

If you're authoring a control that enhances a single constituent control, or is an assembly of constituent controls, your UserControl object will be unable to receive the focus, regardless of the setting of the CanGetFocus property, unless *none* of its constituent controls can receive the focus.

If no constituent controls can receive the focus, and CanGetFocus is True, then your UserControl object will receive the same events a user-drawn control receives. The only thing you need to do with these events is provide a visual indication that your control has the focus.

How Constituent Controls Are Affected by CanGetFocus

As long as your control contains at least one constituent control that can receive the focus, the UserControl object will never receive GotFocus and LostFocus events. In

this situation, the CanGetFocus property takes on a new meaning, more on the order of "can be tabbed to."

The reason for this is that the user may click on a constituent control that can receive the focus, and if this happens, the constituent control will get the focus — regardless of the setting of CanGetFocus.

However, if CanGetFocus is True, the user will not be able to *tab* to any of your control's constituent controls. Generally speaking, this is not a good idea.

EnterFocus and ExitFocus

When the focus moves from outside your control to any of your control's constituent controls, the UserControl object will receive an EnterFocus event. The GotFocus event for the constituent control that receives the focus will be raised *after* the UserControl_EnterFocus event procedure.

As long as the focus remains within your control, the UserControl object's focus-related events will not be raised. As the focus moves from one constituent control to another, however, the appropriate GotFocus and LostFocus events of the constituent controls will be raised.

When the focus moves back outside your control, the last constituent control that had the focus will receive its LostFocus event. When the event procedure returns, the UserControl object will receive its ExitFocus event.

You can use the EnterFocus event to change which constituent control receives the focus. You may wish to do this in order to restore the focus to the constituent control that last had it, rather than simply allowing the first constituent control in your UserControl's tab order to receive the focus, which is the default behavior.

Tip If your control is complex — as for example an Address control with multiple constituent controls — you may be tempted to validate the data in the ExitFocus event. Don't. Instead, give the user a Validate method that can be called in the LostFocus event provided by the container your control instance is placed on. It's good design practice to give the user of your control as much freedom as possible in using events.

29

Tip Generally speaking, it's not a good idea to use MsgBox when you're debugging focus-related events, because the message box immediately grabs the focus. It's a *very* bad idea to use MsgBox in EnterFocus and ExitFocus events. Use Debug.Print instead.

30

Receiving Focus via Access Keys

Avoid hard coding access keys for your control's constituent controls, because access keys permanently assigned to your control in this fashion will limit a user's freedom to choose access keys for her form. In addition, two instances of your control on the same form will have access key conflicts.

"Allowing Developers to Set Access Keys for Your Control," later in this chapter, discusses how you can give the user of your control the ability to set access keys on instances of your control.

Forwarding Focus to the Next Control in the Tab Order

If your control cannot receive the focus itself, and has no constituent controls that can receive the focus, you can give your control the same behavior displayed by Label controls. That is, when the access key for your control is pressed, the focus is forwarded to the next control in the tab order.

To enable this behavior, set the ForwardFocus property of the UserControl object to True.

Controls You Can Use As Constituent Controls

You can place any of the controls supplied with Visual Basic on a UserControl, with the exception of the OLE container control.

Any ActiveX control you've purchased, or any control written to the older OLE specification, can be placed on a UserControl.

As long as you're authoring a control for your own use, that's all you need to know. However, if you're going to distribute your control to others, even if you're giving it away, you need to consider distribution and licensing issues.

Note Toolbox objects other than controls, such as insertable objects — for example, Microsoft Excel Charts — cannot be placed on UserControl objects.

31

The Easy Part — UserControl and Intrinsics

The UserControl object and the Visual Basic intrinsic controls are created by the Visual Basic run-time DLL. Anyone who installs your .ocx file will automatically get a copy of the run-time DLL and support files, so if you author your controls using just the UserControl and intrinsic controls, you have no further licensing or distribution issues to worry about.

The intrinsic controls include: PictureBox, Label, TextBox, Frame, CommandButton, CheckBox, OptionButton, ComboBox, ListBox, HScrollBar, VScrollBar, Timer, DriveListBox, DirListBox, FileListBox, Shape, Line, Image, and Data.

ActiveX controls included with the Professional Edition of Visual Basic are subject to licensing rules, as explained below.

Distributing Constituent Controls

An instance of your control is composed of a UserControl object and its constituent controls, as explained in "The UserControl Object," earlier in this chapter. In order to add an instance of your control to a form, a developer must be able to create these objects.

SetupWizard makes this task easy. When you create a Setup program for your .ocx file, SetupWizard includes all the .ocx files for the constituent controls, along with the Visual Basic run-time DLL and any necessary support files.

When a developer runs your Setup program, the .ocx files that provide the constituent controls are installed on his computer. The only other thing he needs to worry about is whether he has the legal right to use them.

If none of the constituent controls require a license, the developer is set. However, if you used controls you purchased, or any of the ActiveX controls included with Visual Basic, Professional Edition, there are licensing requirements to be met.

Licensing Constituent Controls

When you purchase a control, you generally acquire the right to distribute instances of that control royalty-free as part any application you create. However, such license agreements do *not* give you the right to sell or give away the control to other developers — which is what you're doing when you use it as a constituent control.

So the rule is: In order to use your control, a developer must have licenses for all the licensed controls you've used as constituent controls.

Distribution and Licensing Examples

Applying the rule yields the following examples.

ActiveX Controls Included with Visual Basic

Suppose you author some controls using some of the ActiveX controls included with Visual Basic, Professional Edition. SetupWizard adds the necessary .ocx files to your Setup program.

- A developer who has a copy of Visual Basic, Professional Edition buys your .ocx and installs it. She already has the .ocx files, and the license for them, so she has everything she needs.
- A student who has a copy of Visual Basic, Standard Edition buys your control and installs it. He has the .ocx files, but doesn't have the license to use them.
- A stock market analyst who has a copy of Microsoft Excel buys your control and installs it. She has the .ocx files, but doesn't have the license to use them.

15

ActiveX Controls You've Purchased

Suppose you purchase MegaDino.ocx from Late Cretaceous Computing, and use the Tyrannosaur and Velociraptor controls from this .ocx to develop your own UltimatePredator control. You package this control in UPred.ocx, and you give it away.

Anyone to whom you give a copy of UPred.ocx must have purchased and installed MegaDino.ocx in order to use the UltimatePredator control legally. This is true regardless of the development software they're using.

In fact, if the creators of MegaDino.ocx and DinoRama.ocx used the standard registry key licensing scheme, people to whom you give UPred.ocx will be *unable* to use the UltimatePredator control unless they have MegaDino.ocx installed.

Shareware Controls

Suppose you author your control using a shareware control.

If you sell your control component (.ocx file), the purchaser must also pay the author of the shareware control the appropriate license fee.

If you distribute your control component as shareware, a person who wants to use it must pay the appropriate license fees to you and to the author of the shareware control you used.

Constituent Controls and the Internet

If you want people to be able to use your control on World Wide Web pages, remember that the rule for Web servers is exactly the same as the rule for developers. That is, in order to use your control, a Web server must have licenses for all the licensed controls you've used as constituent controls.

For More Information Licensing issues, including how to add licensing support for the controls you author, how licensing support works, and the mechanism for using licensed controls with the World Wide Web are discussed in "Licensing Issues for Controls," later in this chapter.

32

Object Models for Controls

Complex controls such as TreeView and Toolbar provide run-time access to their functionality through objects. For example, the TreeView control has a Nodes collection containing Node objects that represent the items in the hierarchy the TreeView control displays. Users can create new nodes using the Add method of the Nodes collection.

Objects like Node and Nodes are called *dependent objects.* Dependent objects exist only as a part of some other object, as Node objects are always part of a TreeView control. They cannot be created independently.

You can provide dependent objects like Node and Nodes by including class modules in your ActiveX control project and organizing them into an object model. Object models can be as simple as the Nodes collection with its Node objects, or arbitrarily complex.

Important Control components can only provide dependent objects. They cannot provide objects that can be independently created, using the New operator or the CreateObject function.

33

For More Information Dependent objects are discussed in "Instancing for Classes Provided by ActiveX Components," in "General Principles of Component Design."

Also, you can read about object models in “Organizing Objects: The Object Model,” which is also in “General Principles of Component Design.”

Some design considerations for collections in controls are discussed in “Creating Robust Controls,” later in this chapter. More information on robust techniques for using objects can be found in “Private Communications Between Your Objects,” in “General Principles of Component Design.”

Classes, class modules, and objects are discussed in ”Programming with Objects.”

34

Allowing Developers to Put Controls on Your Control

Some controls can act as containers for other controls. For example, if you place controls on a Visual Basic PictureBox control, all of the controls move when you move the PictureBox. Visual Basic users take advantage of this capability to group controls, produce scrollable pictures, and so on.

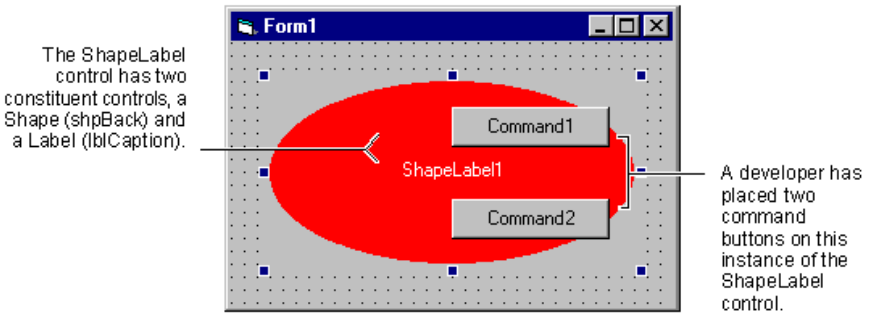
You can allow developers to place controls on your ActiveX control by setting the ControlContainer property of the UserControl object to True.

Controls a developer places on an instance of your ActiveX control can be accessed using the ContainedControls collection of the UserControl object. You can use this collection at either design time or run time.

The ContainedControls Collection vs. the Controls Collection

The ContainedControls collection is different from the Controls collection, which contains only the constituent controls you have used in designing your ActiveX control. This is illustrated in Figure 9.7, which supposes that the ShapeLabel control’s ControlContainer property is True.

Figure 9.7 The Controls and ContainedControls collections



If the control includes the code below, it will always display two names, shpBack and lblCaption.

```
Dim c As Object
For Each c In UserControl.Controls
   MsgBox c.Name
Next
```

If the control includes the code below, it will display Command1 and Command2 — for this instance only.

```
Dim c As Object
For Each c In _
       UserControl.ContainedControls
   MsgBox c.Name
Next
```

16

Availability of the ContainedControls Collection

You cannot access the ContainedControls collection in the Initialize event of your UserControl object. Support for the ControlContainer feature is provided by the object your control is placed on, so your control must be sited on the container object before ContainedControls is available. When the UserControl object receives its ReadProperties event, siting has occurred.

Once your control is sited, and support for the ControlContainer feature is present, the ContainedControls collection may not immediately contain references to the controls a developer has placed on your control. For example, if your control is on a Visual Basic form, the Count property of the ContainedControls collection will be zero until after the UserControl_ReadProperties event procedure has executed.

Performance Impact of ControlContainer

There is extra overhead required to allow a developer to place controls on instances of your ActiveX control. Clipping must be done for the contained controls, which must appear on top of all the constituent controls in your UserControl, and of course the ContainedControls collection must be maintained.

In other words, controls that serve as containers for other controls are heavyweight controls.

For best performance of your controls, you should set ContainedControls to True only if it makes sense for a particular control. For example, it doesn't make much sense for a control assembly like an Address Control to be a container for other controls.

Support for ControlContainer

ControlContainer support will not work for every container your control may be placed on. Visual Basic forms support the *ISimpleFrame* interface that enables the ControlContainer feature, so your control can always support this capability on a Visual Basic form.

If an instance of your control is placed on a container that is not aware of ISimpleFrame, ControlContainer support will be disabled. Your control will continue to work correctly in all other ways, but developers will be unable to place other controls on an instance of your control.

In order for the ContainedControls collection to be available, an ISimpleFrame-aware container must implement the IVBGetControls interface. Calls to the collection will cause errors if the container does not implement this interface, so it's a good idea to use error handling when you access the collection.

Note Controls placed on a container with a transparent background are not visible. If you want your control to be a control container, don't give it a transparent background.

35

Allowing Your Control to be Enabled and Disabled

The Enabled property is an odd beast. It's an extender property, but the Extender object doesn't provide it unless your control has an Enabled property of its own, with the correct procedure ID. If the extender's Enabled property isn't present, your control will not display the same enabled/disabled behavior as other controls.

Your property should delegate to the Enabled property of the UserControl object, as shown in the following code sample:

```
Public Property Get Enabled() As Boolean
    Enabled = UserControl.Enabled
End Property

Public Property Let Enabled(ByVal NewValue As Boolean)
    UserControl.Enabled = NewValue
    PropertyChanged "Enabled"
End Property
```

36

Add this code to the code window of the UserControl your ActiveX control is based on, as discussed in "Adding Properties to Controls," later in this chapter.

Note You can easily add the Enabled property by using the ActiveX Control Interface Wizard to create the interface for your control. The wizard includes the Enabled property in its list of recommended properties.

37

Notice that the Enabled property of the UserControl object is qualified by the object's class name (UserControl). The class name can be used to distinguish properties and methods of the UserControl object from members of your ActiveX control which have the same names.

The Enabled property of the UserControl object acts much like the Enabled property of a form, enabling and disabling the UserControl and all of its constituent controls.

Note The purpose and importance of PropertyChanged are discussed in "Adding Properties to Controls," later in this chapter.

38

Assigning the Procedure ID for the Enabled Property

In order for your Enabled property to work correctly, you need to assign it the Enabled procedure ID. Procedure IDs are discussed in "Properties You Should Provide," later in this chapter.

To assign the procedure ID for the Enabled property

1 On the **Tools** menu, click **Procedure Attributes** to open the **Procedure Attributes** dialog box.

2 In the **Name** box, select your **Enabled** procedure.

3 Click **Advanced** to expand the **Procedure Attributes** dialog box.

4 In the **Procedure ID** box, select **Enabled** to give the property the correct identifier.

17

When you give your Enabled property the correct procedure ID, the container's Extender object shadows it with its own Enabled property; when the user sets the extender property, the container sets your Enabled property.

The reason for this odd arrangement is to ensure consistent Windows behavior. When a form is disabled, it's supposed to disable all of its controls, but the controls are supposed to continue to paint themselves as if they were enabled.

A Visual Basic form conforms to this behavior by tricking its controls. It sets the Extender object's Enabled property to False for all of its controls, *without* calling the Enabled properties of the controls. The controls think they're enabled, and paint themselves so, but in code they appear to be disabled.

If your control has an Enabled property without the Enabled procedure ID, it will remain enabled in code while all the controls around it are disabled. You can see this by putting a command button and a control of your own on a form, and adding the following code:

```
Private Sub Command1_Click()
    Form1.Enabled = False
    Debug.Print Command1.Enabled
    Debug.Print MyControl1.Enabled
End Sub
```

39

Run the program before and after assigning the Enabled procedure ID to your control's Enabled property. In the first case, you'll see that the command button's Enabled property returns False, while your control's Enabled property returns True. After the procedure ID is assigned, both controls will return False.

Correct Behavior for the Enabled Property

You should avoid setting the UserControl's Enabled property, except in your control's Enabled property. The reason for this is that the container is responsible for enabling and disabling the controls it contains. It's rude for a control to tamper with properties the user is supposedly in control of.

Painting a User-Drawn Control's Disabled State

When you author a user-drawn control, you have to provide your own representation of your control's disabled state. If you have implemented an Enabled property as shown above, you can determine when you need to do this by testing the value of UserControl.Enabled in the UserControl_Paint event procedure.

For More Information See "User-Drawn Controls," later in this chapter.

40

Giving Your Control a Transparent Background

Setting the BackStyle property of the UserControl object to Transparent allows whatever is behind your control to be seen, in between the constituent controls on your UserControl's surface.

If one of the constituent controls on the UserControl is a Label whose BackStyle property has also been set to Transparent, and whose Font property specifies a TrueType font, Visual Basic will clip around the font. In addition, mouse clicks that fall in the spaces between letters will be passed through to the container.

Setting BackStyle to Transparent may affect the performance of your control. If your control uses a large number of constituent controls, or a Label control with a transparent background, a TrueType font, and a large amount of text, Visual Basic must do a great deal of clipping to make the background show through correctly.

Note Controls placed on a container with a transparent background are not visible. If you want your control to be a control container, don't give it a transparent background.

41

Allowing Developers to Set Access Keys for Your Control

Placing ampersand characters (&) in the captions of Label controls and CommandButton controls creates access keys with which the end user of an application can shift focus to the control. You can create a similar effect in your ActiveX controls.

For example, suppose you have created a user-drawn button. In the Property Let for your button's Caption property, you can examine the text of the caption the user has entered. If there's an ampersand in front of a letter, you can assign that letter to the AccessKeys property of the UserControl object.

When the end user presses one of the access keys enabled in this fashion, your UserControl object receives an AccessKeyPress event. The argument of this event contains the access key that was pressed, allowing you to support multiple access keys on a control.

Access Keys for Control Assemblies

Control assemblies may contain constituent controls that can get the focus, and that support access keys of their own. You can use this fact to provide access key functionality.

Suppose you've authored a general-purpose control that consists of a text box and a label; you want the user to be able to set an access key in the label's caption, and forward the focus to the text box. You can accomplish this by giving the label and text box TabIndex values of zero and one (the TabIndex values on the UserControl are not visible outside your control), and delegating the Caption property of your control to the label, thus:

```
Property Get Caption() As String
    Caption = Label1.Caption
End Property

Property Let Caption(NewCaption As String)
    Label1.Caption = NewCaption
End Property
```

42

When a developer assigns the text "&Marsupial" to your Caption property, the label control will do all the access key work for you.

Note When the end user presses an access key on one of your constituent controls, the UserControl does *not* receive an AccessKeyPress event.

43

Control Assemblies with Fixed Text

For fixed-purpose control assemblies, such as an Address control, you can put ampersand characters (&) in the captions of constituent controls. Unfortunately, these hard-coded access keys may conflict with other access key choices the user wishes to make on a form.

In a more sophisticated variation of this scheme, you might add an AccessKeyXxxx property to your control for the appropriate constituent controls. That is, if the caption of the label next to the txtLastname control was "Last Name," you would add an AccessKeyLastName property. The developer using your control could assign any character from the label's caption to this property, and in the Property Let code you could change the caption to contain the ampersand.

Making Your Control Align to the Edges of Forms

When creating controls like toolbars and status bars, it's useful to be able to make the control align to one of the borders of the form it's placed on. You can give your

ActiveX control this capability by setting the Alignable property of the UserControl object to True.

If the container your control is placed on supports aligned controls, it will add an Align property to the Extender object. When the user chooses an alignment, your control will automatically be aligned appropriately.

In the Resize event of your UserControl, you will have to redraw your user-drawn control or rearrange the constituent controls of your control assembly. You can use the Align property of the Extender object to determine which container edge the control instance has been aligned to.

Note Not all containers support alignable controls. If you attempt to test the value of the Align property, make sure you use error trapping.

44

Making Your Control Invisible at Run Time

To author a control that's invisible at run time, like the Timer control, set the InvisibleAtRuntime property of the UserControl object to True.

Important Don't use the Visible property of the Extender object to make your control invisible at run time. If you do, your control will still have all the overhead of a visible control at run time. Furthermore, the extender properties are available to the user, who may make your control visible.

45

Invisible Controls vs. Ordinary Objects

Before you create a control that's invisible at run time, consider creating an ordinary object provided by an in-process code component (ActiveX DLL) instead.

Objects provided by in-process code components require fewer resources than controls, even invisible controls. The only reason to implement an invisible control is to take advantage of a feature that's only available to ActiveX controls.

Setting a Fixed Size for Your Control

Controls that are invisible at run time typically maintain a fixed size. You can duplicate this behavior using the Size method of the UserControl object, as shown here:

```
Private Sub UserControl_Resize()
    Size 420, 420
End Sub
```

46

The Width and Height properties of a UserControl object are always given in Twips, regardless of ScaleMode.

Adding an AboutBox to Your Control

ActiveX controls typically have an About “property” at the top of the Properties window, with an ellipsis button. Clicking the button shows an About box identifying the control and the software vendor that created it.

Visual Basic makes it easy to provide such About boxes. You can have separate About boxes for each control in your control component (.ocx file), or one About box that all the controls in the component share.

To add an About box to a control component

5 Create an About box by adding a form to your ActiveX control project, and giving it appropriate text and controls. Name the form dlgAbout.

6 In the code window for any control in the project, add the following Sub procedure:

```
Public Sub ShowAboutBox()
    dlgAbout.Show vbModal
    Unload dlgAbout
    Set dlgAbout = Nothing
End Sub
```

Important Unloading the About box and setting it to Nothing frees the memory it was using. This is a courtesy to the user of your controls.

7 On the **Tools** menu, click **Procedure Attributes** to open the **Procedure Attributes** dialog box. If the ShowAboutBox procedure is not selected in the **Name** box, click the drop down and select it.

8 Click **Advanced** to expand the **Procedure Attributes** dialog box.

9 In the **Procedure ID** box, select **AboutBox** to give the ShowAboutBox procedure the correct identifier.

10 Repeat steps 2 through 5 for each control in the project.

Note The name of the About box form and the method that shows it can be anything you like. The procedure above used dlgAbout and ShowAboutBox for purposes of illustration only.

If you wish to have separate About boxes for each control, simply create additional forms, and show a different form in each control’s ShowAboutBox method.

Of course, each form you add to the project increases its size. You can get the same effect with the single dlgAbout form by giving it a property named, let us say, ControlID. This property identifies which control dlgAbout is being shown for. In each control’s ShowAboutBox method, set the ControlID property before showing dlgAbout. Place code in the Load event of dlgAbout to change the text and bitmaps on the About box appropriately.

For More Information Adding properties and methods to forms is discussed in “Programming with Objects.”

Providing a Toolbox Bitmap for Your Control

The Toolbox bitmap size is 16 pixels wide by 15 pixels high, as specified by the ActiveX control specification. You can create a bitmap this size, and assign it to the ToolboxBitmap property of your UserControl object.

Important Do not assign an icon to the ToolboxBitmap property. Icons do not scale to well to Toolbox bitmap size.

49

Visual Basic automatically uses the class name of your control as the tool tip text when users hover the mouse pointer over your icon in the Toolbox.

Tip When creating bitmaps, remember that for many forms of color-blindness, colors with the same overall level of brightness will appear to be the same. You can avoid this by restricting your bitmap to white, black, and shades of gray, or by careful color selection.

50

Allowing Your Control to be a Default or Cancel Button

Default and cancel buttons are controlled by the container. To notify the container that your control is capable of being a default or cancel button, set the DefaultCancel property of the UserControl to True.

If you have set the Default or Cancel property to True for one of the constituent controls on your UserControl, you must set the DefaultCancel property of the UserControl to True or the constituent control property will be ignored.

User-drawn controls can examine the DisplayAsDefault property of the Ambient object to determine whether they should draw the extra black border that visually identifies a default button.

If DefaultCancel is True, and the user makes an instance of your control the default button, the UserControl's AccessKeyPressed event will occur when the user presses the Enter key. The argument of the event will contain the ASCII value 13.

If an instance of your control is the cancel button, the argument of the AccessKeyPress event will contain the ASCII value 27.

Important The status of a default or cancel button can change at any time. You must place code in the UserControl's AmbientChanged event to detect changes in the DisplayAsDefault property, and adjust your control's appearance accordingly.

51

Adding Internet Features to Controls

ActiveX controls created with Visual Basic can support asynchronous downloading of property values, such as Picture properties that may contain bitmaps. Through the

Hyperlink property of the UserControl object, they can also request that a browser jump to a URL, or navigate through the history list.

These features are available when the control is placed on a container that provides the necessary support functions, such as Microsoft Internet Explorer.

You may wish to design your control to support both normal loading of property values from a PropertyBag, which is not supported by browsers, and asynchronous downloading of property values.

Note Asynchronous downloading of *local files* is available in any application.

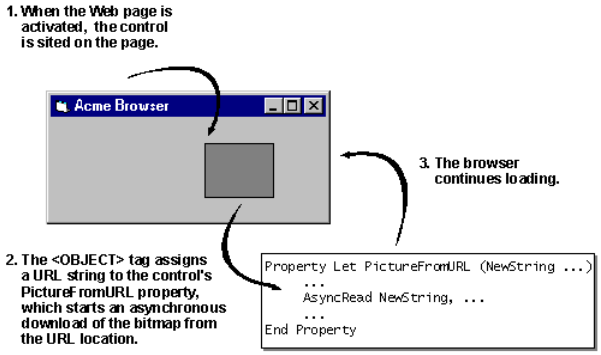
52

Asynchronous Downloading

A control requests asynchronous downloading of a property by calling the AsyncRead method of the UserControl object. This method can be called from any event, method, or property procedure of your control, as long as the control has already been sited on the container.

The call returns immediately, and downloading proceeds in the background, as shown in Figure 9.8a.

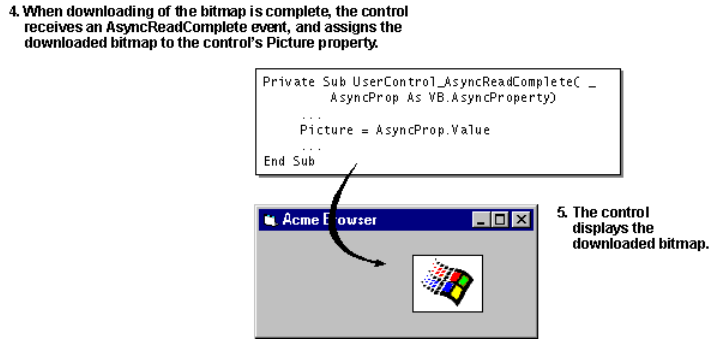
Figure 9.8a Starting asynchronous download of a bitmap property



21

When the container has retrieved the entire property value, the control receives an AsyncReadComplete event that identifies the property whose value has been retrieved. The control can then access the retrieved data and set the property value, as shown in Figure 9.8b.

Figure 9.8b Asynchronous download completes



22

The example code in this topic follows this scheme to create a simple control that displays a bitmap. A more elaborate version of this example is the AsyncPicture control in the CtlPlus.vbp sample application. It has more bells and whistles, but fewer explanations.

To work through the example, open a new Standard EXE project. Use the Project menu to add a UserControl to the project. Place a PictureBox control on the UserControl, and set the properties as shown in the following table:

Object	Property	Setting
UserControl	Name	AsyncBitmap
PictureBox	Name	picBitmap
	AutoSize	True

53

The control will have two properties, an ordinary Picture property and a related PictureFromURL property. The Picture property is implemented with all three property procedures, because a Picture object can be assigned with or without the Set statement.

```
Public Property Get Picture() As Picture
    Set Picture = picBitmap.Picture
End Property

Public Property Let Picture(ByVal NewPicture _
        As Picture)
    Set picBitmap.Picture = NewPicture
    PropertyChanged "Picture"
End Property

Public Property Set Picture(ByVal NewPicture _
        As Picture)
    Set picBitmap.Picture = NewPicture
    PropertyChanged "Picture"
End Property
```

54

The Picture property of the ActiveX control simply delegates to the Picture property of the picture box, picBitmap. The picture box does all the work of displaying the bitmap.

The new property overrides the Picture property of the UserControl object. From now on, if you type Picture without qualifying it, you will get the ActiveX control's Picture property, as defined here. To access the Picture property of the UserControl, you must now qualify it by typing UserControl.Picture.

Note The purpose and importance of PropertyChanged are discussed in "Adding Properties to Controls," later in this chapter.

55

The PictureFromURL Property

When a URL string is assigned to the ActiveX control's PictureFromURL property, the Property Let begins a download of the bitmap. When the download is complete, the bitmap is assigned to the Picture property. PictureFromURL is thus an alternate way of assigning a value to the Picture property.

The PictureFromURL property stores the URL string in a private data member, in the Declarations section of the UserControl:

```
Option Explicit
Private mstrPictureFromURL As String
```

56

The Property Get simply returns this string, so the program can discover where the picture was downloaded from. The Property Let does all the work:

```
Public Property Get () As String
    PictureFromURL = mstrPictureFromURL
End Property

Public Property Let PictureFromURL(ByVal NewString _
        As String)
    ' (Code to validate path or URL omitted.)
    mstrPictureFromURL = NewString
    If (Ambient.UserMode = True) _
            And (NewString <> "") Then
        ' If program is in run mode, and the URL string
        ' is not empty, begin the download.
        AsyncRead NewString, vbAsyncTypePicture, _
            "PictureFromURL"
    End If
    PropertyChanged "PictureFromURL"
End Property
```

57

When a URL string is assigned to the PictureFromURL property, the string is saved in the private variable. If the project the control has been added to is in run mode (which will always be true for a control on an HTML page) and if the URL string is not empty, the Property Let starts an asynchronous download, and then exits.

The asynchronous download is begun by calling the UserControl's AsyncRead method, which has the following syntax:

AsyncRead Target**,** *AsyncType* **[,** *Property***]**

58

The *Target* argument specifies the location of the data. This can be a path or a URL. The host determines the correct method to retrieve the data.

The *AsyncType* argument specifies the form in which the retrieved data will be provided. The Value property of data It has the following possible values:

Constant	Description
vbAsyncTypePicture	The retrieved data is provided as a Picture object.
VbAsyncTypeFile	The retrieved data is placed in a file created by Visual Basic. A string containing the path to the file is provided. This is useful when the data contains a large AVI file to be played. The control author can assign the string to the file name property of the appropriate constituent control.
VbAsyncTypeByteArray	The retrieved data is provided as a byte array. It is assumed that the control author will know how to handle this data.

59

The *Property* argument is a string containing the name of the property whose value is to be downloaded. You can use this to enable multiple simultaneous downloads, because this same string is returned in the AsyncReadComplete event, and can be used in a Select statement.

The value of the Property argument can also be used as the argument of the CancelAsyncRead method, described below.

Completing the Download

As each requested download completes, the control receives an AsyncReadComplete event. The argument of the AsyncReadComplete event is a reference to an AsyncProperty object, which can be used to identify the downloaded property and retrieve the downloaded data.

```
Private Sub UserControl_AsyncReadComplete( _
        AsyncProp As VB.AsyncProperty)
    On Error Resume Next
    Select Case AsyncProp.PropertyName
        Case "PictureFromURL"
            Set Picture = AsyncProp.Value
            Debug.Print "Download complete"
    End Select
End Sub
```

60

You should place error handling code in this event procedure, because an error condition may have stopped the download. If this was the case, that error will be raised when you access the Value property of the AsyncProperty object.

When the downloaded bitmap is assigned to the Picture property, the control repaints.

Note In addition to the Value property that contains the downloaded data and the PropertyName property that contains the name of the property being downloaded, the AsyncProperty object has an AsyncType property. This

contains the same value as the AsyncType argument of the AsyncRead method that started the download.

61

A Bit More Code

So far, the example has no way to save a bitmap assigned at design time. That is, there is no code in the InitProperties, ReadProperties, and WriteProperties events. That code is omitted here, because the purpose of the example is to show asynchronous downloading of a local file.

One more bit of code will prove useful, however. The reason for using a picture box to display the bitmap, instead of simply using the Picture property of the UserControl, is to take advantage of the AutoSize property of the picture box. The following code resizes the entire control whenever the picture box is resized by the arrival of a new bitmap.

```
Private Sub picBitmap_Resize()
    ' If there's a Picture assigned, resize.
    If picBitmap.Picture <> 0 Then
        UserControl.Size picBitmap.Width, _
            picBitmap.Height
    End If
End Sub
```

62

The resizing only happens if the picture box actually contains a Picture object. This allows the user to specify the size of the empty picture box while the download is pending. The code is as follows:

```
Private Sub UserControl_Resize()
    If picBitmap.Picture = 0 Then
        picBitmap.Move 0, 0, ScaleWidth, ScaleHeight
    Else
        If (Width <> picBitmap.Width) _
                Or (Height <> picBitmap.Height) Then
            Size picBitmap.Width, picBitmap.Height
        End If
    End If
End Sub
```

63

If there is no Picture object, the picture box is sized to fill the visible area of the UserControl. If there is a Picture object, and the UserControl is resized, it will snap back to the size of the picture box.

Starting the Download

Close the UserControl designer. The control is now running, even though the rest of the project is in design mode. The default control icon on the Toolbox is enabled, so you can add an instance of the control to Form1.

Locate a large bitmap on your computer — the larger the better. Note the file name and path to the file. Add the following code to the Declarations section of Form1, substituting the name and path of your bitmap for the one shown here:

```
Option Explicit
Const DOWNLOADFILE = "file:\windows\forest.bmp"
```

64

Make sure the string begins with file:\ so that it's a valid URL for a local file.

Place the following code in the form's Load event:

```
Private Sub Form_Load()
    AsyncBitmap1.PictureFromURL = DOWNLOADFILE
    DEBUG.PRINT "Load event complete"
End Sub
```

65

When the form loads, the URL for the local bitmap file will be assigned to the PictureFromURL property, starting the download.

Running the Sample

Press F5 to run the project. In the Immediate window, notice that the first message is "Load event complete," followed by the "Download complete" message from the AsyncReadComplete event.

If the bitmap was large enough, you may also have noticed that Form1 painted while the bitmap was still downloading. Close Form1, to return to design mode.

Run the project again, and this time click the Close button on Form1 as soon as you see it. Because the download is proceeding asynchronously in the background, the form becomes active and can respond to user input before the bitmap has been loaded.

Up to this point, the example only demonstrates downloading of a local file. It can't be used on a Web page, because controls in Standard EXE projects cannot be used by other applications. The AsyncPicture control in the CtlPlus.vbp sample application can actually be used with HTML.

Canceling Asynchronous Downloads

You can call the CancelAsyncRead method to cancel an asynchronous data load. CancelAsyncRead takes the property name as an argument; this must match the value of the PropertyName argument in a prior AsyncRead method call.

Only the specified data load is canceled. All others continue normally.

Navigating with the Hyperlink Object

The Hyperlink object gives your control access to ActiveX hyperlinking functionality. Using the properties and methods of the Hyperlink object, your control can request a hyperlink-aware container, such as Microsoft Internet Explorer, to jump to a given URL or to navigate through the history list.

You can access the Hyperlink object through the Hyperlink property of the UserControl object. The following code fragment assumes that the control's URLText property contains a URL string. Clicking on the control causes it to request that its container navigate to that URL.

```
Private Sub UserControl_Click()
```

```
    HyperLink.NavigateTo Target:=URLText
End Sub
```

66

If the target is not a valid URL or document, an error occurs. If the control is sited on a container that does not support hyperlinking, an application that is registered as supporting hyperlinking is started to handle the request.

The NavigateTo method accepts an optional *Location* argument, which specifies a location within the target URL. If a location is not specified, the server will jump to the top of the URL or document.

The NavigateTo method also accepts an optional *FrameName* argument, which specifies a frame within the target URL.

Note If your control is placed on a container that does not support the IHLink interface, the Hyperlink property of the UserControl object will return Nothing. You should test the property for this value before attempting to use the Hyperlink object.

67

Moving Through the History List

You can call the GoForward and GoBack methods to navigate through the History list. For example:

```
    Hyperlink.GoForward
```

68

If GoForward and GoBack are called when there are no entries in the History list to move forward to, an error occurs. GoForward and GoBack will also raise errors if the container is not hyperlink-aware.

For More Information Details of Internet support for ActiveX controls authored in Visual Basic can be found on the Microsoft Visual Basic Web site. On the Visual Basic Help menu, click Microsoft on the Web, then click Product News.

Information on designing Internet and intranet applications with Visual Basic can be found in *Building Internet Applications.*

69

Designing Controls for Use With HTML

A control on an HTML page is specified using the HTML <OBJECT> and </OBJECT> tags. When the HTML is processed, the control is created and positioned. If the <OBJECT> tag includes any <PARAM NAME> attributes, the property values supplied with those attributes are passed to the control's ReadProperties event using the standard PropertyBag object, as discussed in "UserControl Lifetime and Key Events."

Once the HTML page is active, the control's property values may also be set by scripts attached to events that occur on the page.

Note If there are no <PARAM NAME> attributes other than those that set extender properties (such as Top and Left), the control may receive an

InitProperties event rather than a ReadProperties event. This behavior is dependent on browser implementation, and should not be relied on.

70

The Setup Wizard makes it easy to create an Internet setup for your control, with cabinet (.cab) files that can be automatically downloaded when a user opens an HTML page containing an instance of your control. Support files, such as MSVBVM50.DLL, can be downloaded separately. P-code .ocx files are very compact, so if support files already exist on a user's computer, downloading can be very fast.

Visual Basic controls can support digital signatures, safe initialization, and safe scripting.

Important In order to use a control that includes licensing support on an HTML page, the a licensed copy of the control component must be installed on the Web server that provides the page. This is discussed in "Licensing Issues for Controls," later in this chapter.

71

Making Your Control Safe for Scripting and Initialization on HTML Pages

Code that's downloaded as a result of opening a page on the World Wide Web doesn't come shrink-wrapped, blazoned with a company name to vouch for its reliability. Users may be understandably skeptical when they're asked to okay the download. If you intend for your control to be used on HTML pages, there are several things you can do to reassure users.

- *Digital signatures* create a path to you (through the company that authorized your certificate), in the event that your control causes harm on a user's system. You can incorporate your signature when you use Setup Wizard to create an Internet setup for your control component.
- Marking your control *safe for scripting* tells users that there's no way a script on an HTML page can use your control to cause harm to their computers, or to obtain information they haven't supplied willingly.
- Marking your control *safe for initialization* lets users know there's no way an HTML author can harm their computers by feeding your control invalid data when the page initializes it.

23

This topic explains how to design your control so that when you create your Internet setup, you'll be able to mark your control as safe for scripting and safe for initialization.

Note The default setting for Internet Explorer is to display a warning and to refuse to download a component that has not been marked safe for scripting and initialization.

72

For More Information The latest information on digital signatures, cabinet files, and Internet setup can be found on the Microsoft Visual Basic Web site. On the Visual Basic Help menu, click Microsoft on the Web, then click Product News.

73

Safe for Scripting

When a Web designer places your control on an HTML page, he uses a scripting language such as JavaScript or Visual Basic, Scripting Edition to access the control's properties, invoke its methods, and handle its events. By marking your control as safe for scripting, you're providing an implicit warrantee: "No matter what VBScript or JavaScript code is used, this control cannot be made to harm a user's computer, or to take information the user hasn't volunteered."

As the author of your control, you can be reasonably sure that in normal use it won't destroy data or compromise the security of a user's computer. Once your control is in the hands of a Web designer, however, you have no guarantee that it will be used in the ways you intended.

Keys to Scripting Safety

As an example of a control that's *not* safe for scripting, consider the rich text box. The RichTextBox control has a SaveFile method that can be used to write the contents of the control to a file. A malicious person could write a script that would cause this control to over-write an operating system file, so that the user's computer would malfunction.

What makes the control unsafe is not that it can save information to a file — it's the fact that *the script* can specify the filename. This observation provides the key to creating controls that are safe for scripting. As long as your control doesn't allow a script to specify the source or target for file or registry operations, or make API calls that can be directly controlled by a script, it is probably safe for scripting.

Thus, a control that permits a Web page designer to do any of the following is probably not safe for scripting:

- Create a file with a name supplied by a script.
- Read a file from the user's hard drive with a name supplied by a script.
- Insert information into the Windows Registry (or into an .ini file), using a key (or filename) supplied by a script.
- Retrieve information from the Windows Registry (or from an .ini file), using a key (or filename) supplied by a script.
- Execute a Windows API function using information supplied by a script.
- Create or manipulate external objects using programmatic IDs (for example, "Excel.Application") that the script supplies.

24

The line between safe and unsafe can be a fine one. For example, a control that uses the SaveSetting method to write information to its own registry key doesn't disqualify

itself for safe scripting by doing so. On the other hand, a control that allows the registry key to be specified (by setting a property or invoking a method) is not safe.

A control that uses a temporary file may be safe for scripting. If the name of that temporary file can be controlled by a script, then the control is not safe for scripting. *Even allowing a script to control the amount of information that goes into the temporary file* will make the control unsafe for scripting, because a script could continue dumping information into the file until the user's hard disk was full.

As a final example, a control that uses API calls is not necessarily unsafe for scripting. Suppose, however, that the control allows a script to supply data that will be passed to an API, and doesn't guard against oversize data overwriting memory, or invalid data corrupting memory. Such a control is not safe for scripting.

As an indication of the seriousness of scripting safety, note that VBScript itself does not include methods to access the registry, save files, or create objects.

Choosing Constituent Controls

You might think that using a constituent control that's not safe for scripting would automatically make your ActiveX control unsafe for scripting. This is not necessarily true.

As explained in "Adding Properties to Controls," later in this chapter, the properties and methods of constituent controls do *not* automatically become part of your control's interface. As long as you avoid exposing the properties and methods that make a constituent control unsafe, you can use it without making your own control unsafe.

For example, if you use the RichTextBox as a constituent control, you should not expose its SaveFile method.

Important Do not provide a property that returns a reference to an unsafe constituent control. A script could use this reference to access the properties and methods that make the control unsafe.

74

Documenting Scripting Safety

Determining whether a control is safe is not a trivial exercise. You may find it helpful to record your design decisions that affect safe scripting. A useful exercise is to construct tables containing the following:

- All of your control's public properties, methods, and events.
- All of the files and registry keys accessed, and all API calls used.

25

If there are any dependencies or data transfer between the elements of these two tables, then the control is probably not safe for scripting.

You may wish to have this documentation reviewed by an experienced programmer who understands both ActiveX controls and scripting.

Safe for Initialization

A control marked as safe for initialization carries an implicit guarantee that it will cause no harm no matter how its properties are initialized.

On an HTML page, your control's initial state is set using PARAM NAME attributes with the OBJECT tag that embeds your control on the page. If a malicious Web designer can make your control steal information or otherwise cause harm by placing invalid data in a PARAM NAME attribute, then your control is not safe for initialization.

The best defense against malicious data is to validate each property value that's obtained in your control's ReadProperties event. All the data a Web designer places in PARAM NAME attributes is supplied to your control through the PropertyBag object in the ReadProperties event. (A well-written control should perform this kind of validation anyway, to prevent problems caused by developers who manually edit .frm files.)

For More Information The most up-to-date information on authoring controls for the Internet can be found on the Microsoft Visual Basic Web site. On the Visual Basic Help menu, click Microsoft on the Web, then click Product News.

75

Using Show and Hide Events

The Show and Hide events can be very useful on Web pages. If your control is performing a resource-intensive task, such as showing a video clip or repeatedly downloading and displaying a stock value, you may want to pause this activity when the Hide event occurs.

The Hide event means that the user has moved on to another page, relegating the page your control is on to the History list. The Show event means that the user has returned to your page, and can thus be the signal for resuming resource-intensive display tasks.

For More Information The Show and Hide events are discussed in "UserControl Lifetime and Key Events," earlier in this chapter.

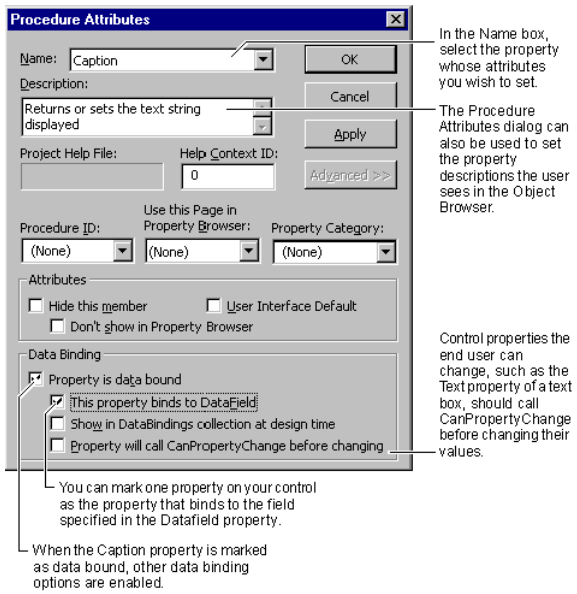
76

Binding a Control to a Data Source

Visual Basic allows you to mark properties of your control as *bindable*. A developer can associate bindable properties with database fields, making it easier to use your control in database applications.

Use the Procedure Attributes dialog box, accessed from the Tools menu, to mark properties of your control as bindable. Figure 9.9 shows the data binding options made available by clicking the dialog's Advanced button.

Figure 9.9 Data binding options for ActiveX control properties



26

The controls supplied with Visual Basic can be bound to database fields using their DataSource and DataField properties. You can select one property of your control to be bound to the DataField property. Typically, this will be the most important piece of data your control holds.

Although you can mark only one field as bound to the field specified in the DataField property, you can mark additional properties of your ActiveX control as bindable. Developers can use the DataBindings collection to bind these additional bindable properties to data fields.

Note Some development tools and control containers do not support data binding. This topic describes the support for data-bound controls provided by Visual Basic.

77

The DataBindings Collection

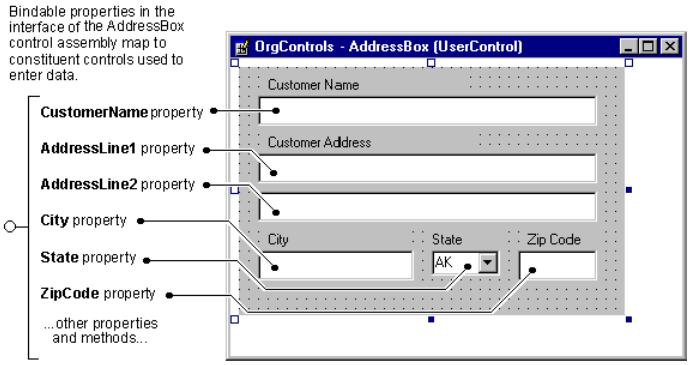
The DataBindings collection is an extender property that Visual Basic provides to users of your control. It allows the developer to access the list of bindable properties on your control.

Note All bindable properties appear in the DataBindings collection at run time. At design time, only properties marked “Show in DataBindings collection at design time” will appear when the DataBindings property is accessed in the Properties window.

78

For example, you might create an Address control assembly, using labels and text boxes as constituent controls. The bindable properties of your control would correspond to the text boxes on your control, as shown in Figure 9.10.

Figure 9.10 An Address control assembly with multiple fields



27

The mapping between properties of the control and contents of the constituent controls is accomplished by delegation, as in this code fragment:

```
Public Property Get AddressLine1() As String
    AddressLine1 = txtAddressLine1.Text
End Property

Public Property Let AddressLine1(NewValue As String)
    If CanPropertyChange("AddressLine1")
        txtAddressLine1.Text = NewValue
        ' The following line tells Visual Basic the
        ' property has changed--if you omit this line,
        ' the data source will not be updated!
        PropertyChanged "AddressLine1"
    End If
End Property
```

79

Delegating to the text box control means that the text box does all the work of displaying the value and accepting user changes. Because the user can change the value of the property while the text box is displaying it, you must also mark the property as changed in the text box's Change event, as shown below.

```
Private Sub txtAddressLine1_Change()
    PropertyChanged "AddressLine1"
End Sub
```

80

Important In order for the new value to be written back to the data source, you must call PropertyChanged. If you don't call the PropertyChanged method, your control will not be bound for update.

81

For More Information The PropertyChanged method has another important purpose, as discussed in "Adding Properties to Controls," later in this chapter.

82

Calling CanPropertyChange

Your control should always call CanPropertyChange before changing the value of a property that can be data-bound. Do not set the property value if CanPropertyChange returns False. Doing so may cause errors in some control containers.

If your control always calls CanPropertyChange, you can check “Property will call CanPropertyChange before changing” on the Procedure Attributes dialog box.

Note At present, CanPropertyChange always returns True in Visual Basic, even if the bound field is read-only in the data source. This does not cause a problem with the code shown above, because Visual Basic doesn’t raise an error when your program attempts to change a read-only field; it just doesn’t update the data source.

83

Discovering and Setting Bindable Properties at Run Time

If a developer placed an instance of the AddressBox control on a form, she could execute the following code to list the bindable properties:

```
Dim dtb As DataBinding
For Each dtb In AddressBox1.DataBindings
    Debug.Print dtb.PropertyName
Next
```

84

At run time, the developer could use the following code to bind the AddressLine1 property to the AddrLine1 field, assuming that field was available on the data source specified by the DataSource extender property:

```
AddressBox1.DataBindings( _
    "AddressLine1").DataField = "AddrLine1"
```

85

Finding Out Whether a Field has Changed

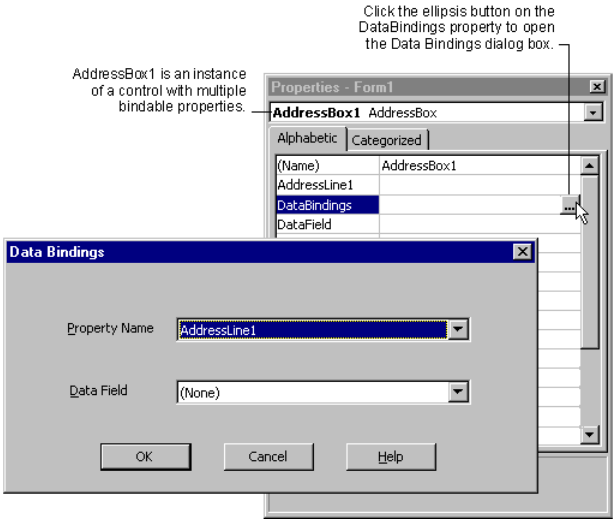
You can test the DataChanged property of a DataBinding object to find out if the value of a field has changed. This property functions in the same way as the DataChanged extender property of bound controls.

Setting Multiple Data Bindings at Design Time

Bindable properties always appear in the DataBindings collection at run time. By default, they do not appear in the Data Bindings dialog box at design time.

If you want a bindable property to appear in the Data Bindings dialog box, select that property in the Procedure Attributes dialog box and check “Show This Property in the Bindings Collection.”

Figure 9.11 Using the Data Bindings dialog box



28

The Data Field box shows all fields available on the data source specified by the current value of the DataSource extender property on the control instance.

Attributes and Flags

If you have developed OLE controls in the past, you can use the following table to see what flags are set by the Procedure Attributes dialog box. The table also shows how these attributes are accessed through the Member object in the Visual Basic Extensibility Model.

Attribute	Flag	Member Object
Property is data bound.	BINDABLE	Bindable
This property binds to DataField.	DEFAULTBIND	DefaultBind
Show in DataBindings collection at design time.	DISPLAYBIND	DisplayBind
Property will call CanPropertyChange before changing.	REQUESTEDIT	RequestEdit

86

Allowing Developers to Edit Your Control at Design Time

Some controls, like the OLE control supplied with Visual Basic, allow you to edit the control's contents at design time. You can enable this behavior for your control by setting the EditAtDesignTime property of your UserControl object to True.

If a container supports this feature, it will add an Edit menu item to the context menu that appears when a developer right-clicks on an instance of your control at design-time.

The developer using the control can activate a control instance by right-clicking the control to get the Context menu, then clicking Edit. The control will be activated, and will behave as it does at run time.

This allows you to author controls with visual design features, such as letting the user size rows and columns, or set property values by typing directly into constituent controls. You can detect whether your control is running at design time by testing the UserMode property of the Ambient object. This property is False at design time.

The control only remains active while it is selected. When the developer clicks on another control, the control will deactivate. To reactivate the control, the developer must select Edit from the context menu.

Note When your control is activated in this fashion, the events of the UserControl object will occur, so that your control can operate normally, but your control will be unable to raise any events. The RaiseEvent method will simply be ignored; it will not cause an error.

87

Drawing Your Control

The way you draw your control's appearance depends on the control creation model you're using.

If you're creating a user-drawn control, you have to do all the drawing yourself. You have to know when to draw your control, what state it's in (for example, clicked or unclicked), and whether you should draw a focus rectangle.

If you're enhancing an existing control or creating a control assembly, on the other hand, your UserControl's constituent controls provide your control's appearance. The constituent controls draw themselves automatically, and all you have to worry about is whether they're positioned correctly on the UserControl.

For More Information Control creation models are discussed in "Control Creation Basics," earlier in this chapter.

88

User-Drawn Controls

When you're doing your own drawing, the only place you need to put drawing code is in the UserControl_Paint event procedure. When the container repaints the area your control is located on, the UserControl object will receive a Paint event.

If the built-in graphics methods of the UserControl object, such as Line and Circle, don't meet your drawing needs, you can use Windows API calls to draw your control. Regardless of the drawing technique, the code goes in UserControl_Paint.

If your control has to change its appearance based on user actions, such as when the user clicks on the control, you can raise the Paint event by calling the UserControl's Refresh method.

Note Do not set BackStyle = Transparent when authoring a user-drawn control. When the UserControl is transparent, no Paint events will be raised and graphics methods will not work.

89

Important In Paint events, do not use DoEvents, or any other code that yields to other programs. Doing so will cause errors.

90

The following example demonstrates the basic principle of a three-state button, each of the three states being represented by a different bitmap. To work through the example, open a new Standard EXE project and use the Project menu to add a UserControl to the project. Place a PictureBox on the UserControl, and set object properties as follows:

Object	Property	Setting
UserControl	Name	TripleState
PictureBox	AutoRedraw	True
	Name	picStates
	Picture	(Any bitmap)
	Visible	False

91

Note The example works best if the bitmap chosen for the Picture property changes color dramatically from left to right.

92

Add the following code to the Declarations section of the UserControl's code window.

```
Option Explicit
' Private variable keeps track of the current state.
Private mintState As Integer
```

93

A simple mechanism is used to provide the three states: Clicking on the control rotates through the states by incrementing the value of a private state variable each time the control is clicked. If the value becomes too large, it is reset to zero. Add this code to the UserControl's Click event.

```
Private Sub UserControl_Click()
    mintState = mintState + 1
    If mintState > 2 Then mintState = 0
    ' The following line causes Paint event to occur.
    Refresh
End Sub
```

94

Add the following code to the UserControl's Paint event. When the Paint event occurs, the code copies one-third of the bitmap in the invisible PictureBox to the

UserControl. Which third is copied depends on the current value of the private state variable mintState.

```
Private Sub UserControl_Paint()
    PaintPicture picStates.Picture, 0, 0, ScaleWidth, _
        ScaleHeight, mintState * picStates.Width / 3, _
        0, picStates.Width / 3
End Sub
```

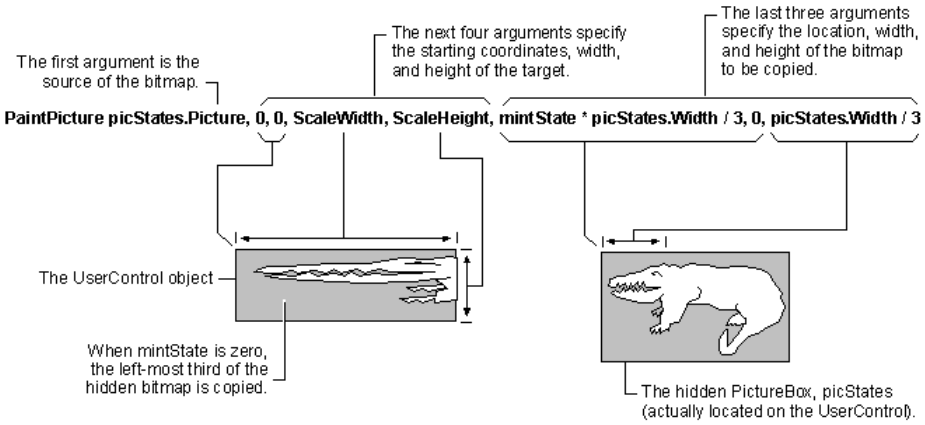
95

Note Another way to provide a different appearance for the each of your control's states is to use the Select…Case statement.

96

When mintState is zero, its initial value, the first third of the hidden bitmap will be copied onto the UserControl, as shown in Figure 9.12.

Figure 9.12 Copying the first third of the hidden bitmap to the UserControl



29

Click the Close box on the UserControl designer, to enable its icon in the Toolbox. Double-click the icon to place a copy of the control on a form, and press F5 to run the project. Click on the control to change the state.

You can hide the form behind another window, or minimize and restore the form, to verify that the control correctly retains its state, and repaints accordingly.

Tip For better performance when drawing your own control, make sure the AutoRedraw property is set to False.

97

For More Information See "Drawing the ShapeLabel Control," in "Creating an ActiveX Control."

Using Windows API calls is discussed in "Accessing the Microsoft Windows API." The PaintPicture method is discussed in "Working with Text and Graphics."

98

Working with Other Events

In similar fashion, you can simulate other event-driven appearance changes, such as button presses. To animate button presses, put state-changing code in the MouseUp and MouseDown events. Regardless of the events being used, the principle is the same: change the state, and call the Refresh method.

Showing That a User-Drawn Control Has the Focus

If your control can get the focus, you will need a second state variable to keep track of whether your control currently has the focus, so that each time your control redraws itself it will show (or not show) an appropriate indication that it has the focus.

The Windows API DrawFocusRect can be used to draw the type of single-pixel dotted line used to show focus in the CommandButton control. There is no comparable API for non-rectangular shapes.

For More Information See "How to Handle Focus in your Control," earlier in this chapter.

99

Showing a User-Drawn Control as Disabled

If you implement an Enabled property, you will need to keep track of whether your control is enabled, so you can provide a visual cue to the user when your control is disabled.

If you implement your control's Enabled property as discussed in "Allowing Your Control to be Enabled and Disabled," earlier in this chapter, you can simply test UserControl.Enabled to determine whether to draw your control as enabled or disabled.

The way you draw your control to indicate that it's disabled is entirely up to you.

User-Drawn Controls That Can Be Default Buttons

Setting the DefaultCancel property of your UserControl object to True tells the container your control can be a default or cancel button, so the Extender object can show Boolean Default and Cancel properties.

You can examine the value of the DisplayAsDefault property of the Ambient object to determine whether your control should show the extra black border that tells the end user your control is the default button. Show the border only when the DisplayAsDefault property is True.

Important Correct behavior for a button in Windows is to show the default border only if your control has been designated as the default button, *and* no other button has the focus. DisplayAsDefault is True only if both of these conditions are met. Other methods of determining when to display the border may result in incorrect behavior.

100

For More Information See “Understanding the Container’s Extender Object,” and “Using the Ambient Object to Stay Consistent with the Container,” earlier in this chapter.

The VBButton sample application demonstrates a button control written using Visual Basic.

101

Providing Appearance Using Constituent Controls

If you’re enhancing an existing control, that single constituent control will typically occupy the entire visible surface of your UserControl object. You can accomplish this by using the Move method of the constituent control in the Resize event of the UserControl, as shown below.

```
Private Sub UserControl_Resize()
    picBase.Move 0, 0, ScaleWidth, ScaleHeight
End Property
```

102

The code above assumes that an enhanced picture control is being authored. A PictureBox control has been placed on the UserControl, and named picBase.

If the control you’re enhancing has a minimum size, or a dimension that increases in large increments — such as the height of a ListBox control, which changes by the height of a line of text — you will have to add code to determine whether the Move method has produced the desired result.

You can rely on the control you’re enhancing to handle painting, including (where appropriate) default button highlighting.

Tip You may also have to add code to resize your UserControl object, to accommodate a constituent control that can’t be sized arbitrarily — such as a text box or list box. To avoid exhausting stack space with recursive calls to the Resize event, use static variables to determine when the UserControl_Resize event procedure is making recursive calls.

103

Resizing a Control Assembly

If you’re authoring a control assembly, the Resize event will be more complex, because you have to adjust both size and relative location of multiple constituent controls.

Enforcing a Minimum Control Size

If you author a control with a large number of constituent controls, there may be a minimum size below which resizing the constituent controls is futile, because too little of each control is visible to be of any use, or because the enforced minimum sizes of some constituent controls has been reached.

There is no real harm in allowing the user to reduce the size of your control to absurdity. Most controls allow this, because preventing it is a lot of work, and because

at some point you have to rely on your user's desire to produce an application that works and is usable.

In the event that resizing below some threshold causes your control to malfunction, you might make all of your constituent controls invisible, as an alternative to enforcing a minimum size.

The following code fragment provides a simple example of enforcing a minimum size.

```
Private Sub UserControl_Resize()
    Dim sngMinH As Single
    Dim sngMinW As Single

    ' Code to resize constituent controls. It is
    ' assumed that each of these will have some minimum
    ' size, which will go into the calculation of the
    ' UserControl's minimum size.

    sngMinW = <<Width calculation>>
    sngMinH = <<Height calculation>>

    If Width < sngMinW Then Width = sngMinW
    If Height < sngMinH Then Height = sngMinH
End Sub
```

104

Notice the <<pseudo-code placeholders>> for the calculation of your control's minimum width and height. These calculations will be in the ScaleMode units of your UserControl. They may be very complicated, involving the widths and heights of several of the constituent controls.

The Width and Height properties of the UserControl are then set, if necessary.

Important The Width and Height properties of the UserControl include the thickness of the border, if any. If BorderStyle = 1 (Fixed Single), the area available for constituent controls will be reduced by two *pixels* (not Twips) in both width and height. If you have exposed the BorderStyle property for your user to set, include code to test the current value.

105

As an alternative, you could use the Size method:

```
    If Width > sngMinW Then sngMinW = Width
    If Height > sngMinH Then sngMinH = Height
    If (sngMinW <> Width) Or (sngMinH <> Height) Then
        ' (Optionally, set recursion flag.)
        Size sngMinW, sngMinH
        ' (Clear recursion flag, if set.)
    End If
```

106

This code is more slightly more complicated, but it simplifies things if you need to avoid recursion when resizing your control, as discussed below.

Important The Width and Height properties of the UserControl are always expressed in Twips, regardless of the ScaleMode setting. If you have set ScaleMode to something other than Twips, use the ScaleX and ScaleY methods to convert your minimum size calculations to Twips.

107

Dealing with Recursion

No code for recursion is included in the example above, and recursion is virtually guaranteed. For example, if you attempt to resize the control so that both width and height are below the minimum values, the Resize event will reset the Width property, which will cause a second Resize to be raised immediately.

This second Resize event will test and reset the height, and then return — so that by the time the first Resize event tests the height, it will already have been reset to the minimum. Clearly, this can lead to confusing debugging situations.

Even if you use the Size method, a second Resize event will occur, repeating all your calculations. This can be avoided by setting a flag when you deliberately resize the control. The Resize event should check this flag, and skip all processing when it is True.

A recursion flag is not necessary for simple minimum size situations, but is virtually required for more complicated scenarios.

For example, if you use the code above in a control whose Align property is True, so that it aligns to the form it's placed on, as described in "Making Your Control Align to the Edges of Forms," infinite recursion errors are guaranteed, until stack space is exhausted and an error occurs.

Important Always use error handling in Resize event procedures. Errors here cannot be handled by the container, and your control component will therefore fail, causing the application using your control to fail, as well.

108

For More Information The models for creating controls are discussed in "Three Ways to Build ActiveX Controls" earlier in this chapter.

109

Adding Properties to Controls

You implement properties of your ActiveX control by adding property procedures to the code module of the UserControl that forms the basis of your control class.

By default, the only properties your control will have are the extender properties provided by the container. You must decide what additional properties your control needs, and add code to save and retrieve the settings of those properties.

Properties for controls differ in two main ways from properties of other objects you create with Visual Basic.

- Property values are displayed in the Properties window and Properties Pages dialog box at design time.
- Property values are saved to and retrieved from the container's source file, so that they persist from one programming session to the next.

30

As a result of these differences, implementing properties for controls has more requirements and options than for other kinds of objects.

Implement Control Properties Using Property Procedures

The most important consequence of the differences listed above is that control properties should always be implemented using property procedures instead of public data members. Otherwise, your control will not work correctly in Visual Basic.

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 at the end of every successful Property Let or Property Set, as shown in the following code fragment.

```
Private mblnMasked As Boolean

Public Property Get Masked() As Boolean
    Masked = mblnMasked
End Property

Public Property Let Masked(ByVal NewValue As Boolean)
    mblnMasked = NewValue
    PropertyChanged "Masked"
End Property
```

110

There are two reasons for notifying Visual Basic that a property value has changed:

- If you don't call PropertyChanged, Visual Basic cannot mark control instances as needing to be saved. They will not get WriteProperties events, and developers who use your control will lose any property values they set at design time.
- Because property values may be displayed in more than one place, the development environment must be informed when a property value changes, so it can synchronize the values shown in the Properties window, the Property Pages dialog box, and so forth.

31

Run-Time Properties

Properties that are available only at run time don't need to call the PropertyChanged method, unless they can be data-bound. However, they still need to be implemented using property procedures, as explained in the related topic "Creating Design-Time-Only or Run-Time-Only Properties."

For More Information Authoring controls that can be bound to data sources is discussed in "Binding Your Control to a Data Source," earlier in this chapter.

111

Properties You Don't Need to Implement

Right away, you can avoid a lot of work. The Extender object, which is provided by the container your control is placed on, will supply a number of properties for you. DragIcon, HelpContextID, TabIndex, Top, and Visible are a few of the extender properties supplied by Visual Basic forms.

For More Information To see all the properties Visual Basic's Extender object provides, search the Language Reference in the Books Online index for Extender Object. The Extender object is discussed in "Understanding the Container's Extender Object," earlier in this chapter. An odd exception is the Enabled property, which you must implement so that the Extender object can mask it. See "Allowing Your Control to be Enabled and Disabled," later in this chapter.

112

For More Information The UserControl object is discussed in "The UserControl Object," earlier in this chapter. General information on creating properties for objects, such as making a property the default for an object, is provided in "Adding Properties and Methods to Classes," in "General Principles of Component Design."

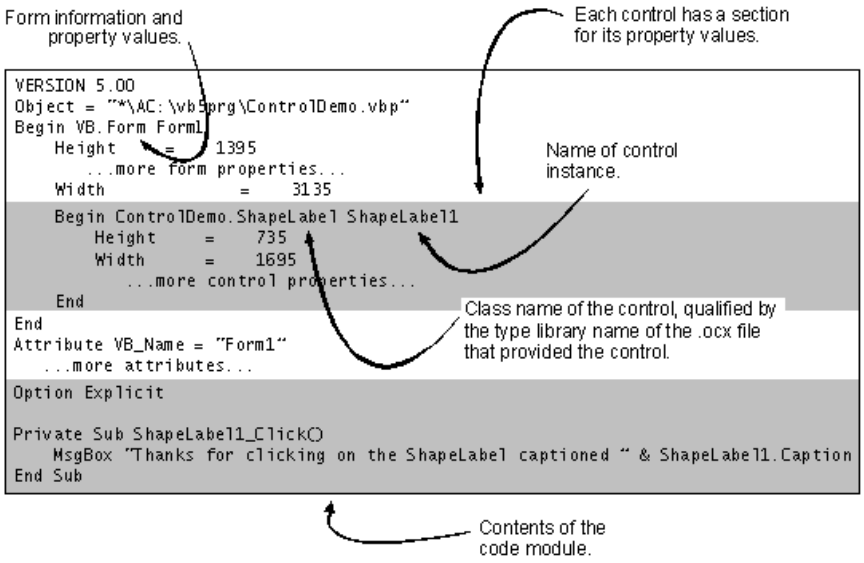
113

Saving the Properties of Your Control

As discussed in "Understanding Control Lifetime and Key Events," earlier in this chapter, instances of controls are continually being created and destroyed — when form designers are opened and closed, when projects are opened and closed, when projects are put into run mode, and so on.

How does a property of a control instance — for example, the Caption property of a Label control — get preserved through all this destruction and re-creation? Visual Basic stores the property values of a control instance in the file belonging to the container the control instance is placed on; .frm/.frx files for forms, .dob/.dox files for UserDocument objects, .ctl/.ctx files for UserControls, and .pag/.pgx files for property pages. Figure 9.13 illustrates this.

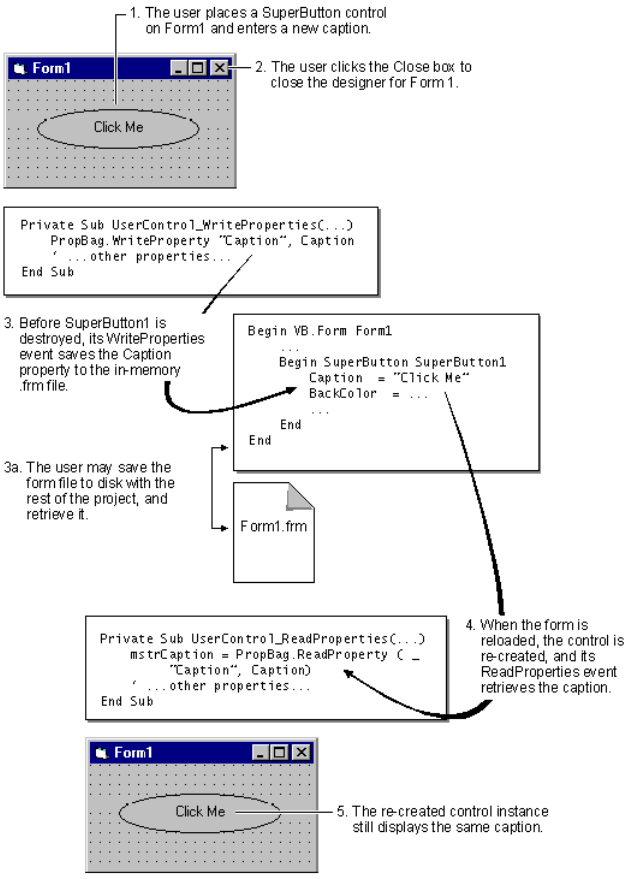
Figure 9.13 An .frm file contains saved control properties



32

When you author a control, you must include code to save your property values before a control instance is destroyed, and read them back in when the control instance is re-created. This is illustrated in Figure 9.14.

Figure 9.14 Saving and retrieving property values



33

Figure 9.14 is slightly oversimplified. You don't actually have to close a form to cause the WriteProperties event procedures of control instances to be executed. Saving a form file to disk causes WriteProperties events to be raised for all controls on the form.

Tip This topic explains the mechanism for saving and retrieving property values in code, but you won't normally have to write all the code described here. The ActiveX Control Interface Wizard can generate most of the code to save and retrieve your property values.

Saving Property Values

Use the PropertyBag object to save and retrieve property values. The PropertyBag is provided as a standard interface for saving property values, independent of the data format the container uses to save its source data.

The following code example uses the Masked property, a Boolean property described in the related topic “Adding Properties to Controls.”

```
Private Sub UserControl_WriteProperties(PropBag As _
        PropertyBag)
    ' Save the value of the Boolean Masked property.
    PropBag.WriteProperty "Masked", Masked, False
    ' . . . more properties . . .
End Sub
```


The WriteProperty method of the PropertyBag object takes three arguments. First is a string that identifies the property being saved, followed by value to be saved — usually supplied by accessing the property, as shown above.

The last parameter is the default value for the property. In this case, the keyword False is supplied. Typically, you would create a global constant, such as PROPDEFAULT_MASKED, to contain this value, because you need to supply it in three different places, in the WriteProperties, ReadProperties, and InitProperties event procedures.

The Importance of Supplying Defaults

It may seem strange, at first, to be supplying a default property value when you’re *saving* the value of a property. This is a courtesy to the user of your control, because it reduces the size of the .frm, .dob, .pag, or .ctl file belonging to the container of the control.

Visual Basic achieves this economy by writing out a line for the property only if the value is *different* from the default. Assuming that the default value of the Masked property is False, the WriteProperty method will write a line for the property only if the user has set it to True.

You can easily see how this technique reduces the size of .frm files by opening a new Standard EXE project, adding a CommandButton to Form1, and saving Form1.frm. Use a text editor such as Notepad or Wordpad to open Form1.frm, and compare the number of properties that were written to the file for Command1 to the number of properties in the Properties window for Command1.

Wherever possible, you should specify default values for the properties of your control when initializing, saving, and retrieving property values.

Retrieving Property Values

Property values are retrieved in the ReadProperties event of the UserControl object, as shown below.

```
Private Sub UserControl_ReadProperties(PropBag As _
        PropertyBag)
    On Error Resume Next
    ' Retrieve the value of the Masked property.
    Masked = PropBag.ReadProperty("Masked", False)
    ' . . . more properties . . .
End Sub
```

116

The ReadProperty method of the PropertyBag object takes two arguments: a string containing the name of the property, and a default value.

The ReadProperty method returns the saved property value, if there is one, or the default value if there is not. Assign the return value of the ReadProperty method to the property, as shown above, so that validation code in the Property Let statement is executed.

If you bypass the Property Let by assigning the property value directly to the private data member or constituent control property that stores the property value while your control is running, you will have to duplicate that validation code in the ReadProperties event.

Tip Always include error trapping in the UserControl_ReadProperties event procedure, to protect your control from invalid property values that may have been entered by users editing the .frm file with text editors.

117

Properties that are Read-Only at Run Time

If you create a property the user can set at design time, but which is read-only at run-time, you have a small problem in the ReadProperties event. You have to set the property value *once* at run time, to the value the user selected at design time.

An obvious way to solve this is to bypass the Property Let, but then you have no protection against invalid property values loaded from source files at design time. The correct solution to this problem is discussed in "Creating Design-Time-Only, Run-Time-Only, or Read-Only Run-Time Properties."

Initializing Property Values

You can assign the initial value of a property in the InitProperties event of the UserControl object. InitProperties occurs only once for each control instance, when the instance is first placed on a container.

Thereafter, as the control instance is destroyed and re-created for form closing and opening, project unloading and loading, running the project, and so on, the control instance will only receive ReadProperties events. This is discussed in "Understanding Control Lifetime and Key Events," earlier in this chapter.

Be sure to initialize each property with the same default value you use when you save and retrieve the property value. Otherwise you will lose the benefits that defaults provide to your user, described in "The Importance of Supplying Defaults," earlier in this topic.

Tip The easiest way to ensure consistent use of default property values is to create global constants for them.

118

Exposing Properties of Constituent Controls

By default, the properties of the UserControl object — and the constituent controls you add to it — are not visible to the end user of your control. This gives you total freedom to determine your control's interface.

Frequently, however, you will want to implement properties of your control by simply delegating to existing properties of the UserControl object, or of the constituent controls you've placed on it. This topic explains the manual technique of exposing properties of the UserControl object or its constituent controls.

Understanding delegation and property mapping will help you get the most out of the ActiveX Control Interface Wizard, which is designed to automate as much of the process as possible. It will also enable you to deal with cases that are too complicated for the wizard to handle.

Exposing Properties by Delegating

Suppose you want to create a control that allows the end user to edit a field with a special format, such as a Driver's License Number. You start by placing a single text box on a UserControl, and naming it something catchy like txtBase.

Because your new control is an enhancement of a single Visual Basic control, you also resize txtBase to fit the UserControl. You do this in the UserControl's Resize event, as discussed in "Providing Appearance Using Constituent Controls," earlier in this chapter.

To create the BackColor property of your control, you can simply expose the BackColor property of the text box, as shown in this code fragment from the UserControl's code module.

```
Public Property Get BackColor() As OLE_COLOR
    BackColor = txtBase.BackColor
End Property

Public Property Let BackColor(ByVal NewColor _
        As OLE_COLOR)
    ' . . . property validation code . . .
    txtBase.BackColor = NewColor
    PropertyChanged "BackColor"
End Property
```

119

The purpose and importance of PropertyChanged are discussed in "Adding Properties to Controls," earlier in this chapter.

The BackColor property you create for your control simply saves its value in the BackColor property of the text box control. Methods are exposed in similar fashion,

by delegating the work to the corresponding method of the control you're enhancing. Delegation is discussed in "Composing Objects," in "Programming with Objects."

Tip When you use the OLE_COLOR data type for color properties, the Properties window will automatically show the ellipsis button that brings up the standard color selection dialog box. Standard property types are discussed in the related topic "Using Standard Control Property Types."

120

Important Because properties of the UserControl object and constituent controls are exposed by delegation, you cannot expose design-time-only properties such as Appearance and ClipControls. The settings you choose for such properties will be fixed for your ActiveX control.

121

Mapping Properties to Multiple Controls

Frequently you will want to map more than one constituent control property to a property of your control. Delegation gives you the flexibility to handle this situation.

Suppose, for example, that you've created your control's appearance by placing several check boxes, option buttons, and labels on a UserControl. It makes sense for the BackColor of your UserControl to be the background color of your control. However, it also makes sense for the BackColor properties of the constituent controls to match this color.

The following code fragment illustrates such an implementation:

```
Public Property Get BackColor() As OLE_COLOR
    BackColor = UserControl.BackColor
End Property

Public Property Let BackColor(ByVal NewColor _
        As OLE_COLOR)
    Dim objCtl As Object
    ' . . . property validation code . . .
    UserControl.BackColor = NewColor
    For Each objCtl In Controls
        If (TypeOf objCtl Is OptionButton) _
                Or (TypeOf objCtl Is CheckBox) _
                Or (TypeOf objCtl Is Label) _
            Then objCtl.BackColor = NewColor
    Next
    PropertyChanged "BackColor"
End Property
```

122

When the property value is read, the value is always supplied from the BackColor property of the UserControl. Always choose a single source to be used for the Property Get.

Note When you give your control a property which the underlying UserControl object already possesses, using that property name in your code will refer to the new property you have defined, unless you qualify the property name with UserControl, as shown above.

123

For More Information Using the Controls collection to iterate over all constituent controls is discussed in “Control Creation Terminology,” earlier in this chapter. The purpose and importance of PropertyChanged are discussed in “Adding Properties to Controls,” earlier in this chapter.

124

Multiple BackColor Properties

The implementation above raises an interesting question. What if you want to provide the user of your control with a way to set the background color of all the text boxes on your control? You’ve already mapped BackColor to its most natural use, but you can always get creative with property names.

For example, you might add a TextBackColor property, modeled on the example above, that would set the BackColor properties of all the text boxes on your control. Choose one text box as the source of the TextBackColor, for the Property Get, and you’re in business. (It doesn’t make much sense to use the UserControl’s BackColor for this purpose.)

Mapping to Multiple Object Properties

As another example of multiple property mapping, you might implement TextFont and LabelFont properties for the control described above. One property would control the font for all the labels, and the other for all the text boxes.

When implementing multiple mapped object properties, you can take advantage of multiple object references. Thus you might implement the LabelFont property as shown in the following code fragment:

```
Public Property Get LabelFont() As Font
    Set LabelFont = UserControl.Font
End Property

' Use Property Set for object properties.
Public Property Set LabelFont(ByVal NewFont As Font)
    Set UserControl.Font = NewFont
    SyncLabelFonts
    PropertyChanged "LabelFont"
End Property

Private Sub SyncLabelFonts()
    Dim objCtl As Object
    For Each objCtl In Controls
        If TypeOf objCtl Is Label Then
            Set objCtl.Font = UserControl.Font
        End If
    Next
End Sub
```

125

The code in the SyncLabelFonts helper function assigns to each Label control’s Font property a reference to the UserControl object’s Font object. Because all of the

controls have references to the same Font object, changes to that font will be reflected in all the labels.

A helper function is used because the same code must be executed when your control is initialized, and when saved properties are read.

Note The purpose and importance of PropertyChanged are discussed in "Adding Properties to Controls," earlier in this chapter.

126

The code to initialize, save, and retrieve the LabelFont property is shown below. Optionally, you can set the characteristics of the UserControl's font to match those of the container's font, as discussed in "Using the Ambient Object to Stay Consistent with the Container."

```
Private Sub UserControl_InitProperties()
    SyncLabelFonts
End Sub

Private Sub UserControl_ReadProperties(PropBag As _
        VB.PropertyBag)
    On Error Resume Next
    Set LabelFont = PropBag.ReadProperty("LabelFont")
End Sub

Private Sub UserControl_WriteProperties(PropBag As _
        VB.PropertyBag)
    PropBag.WriteProperty "LabelFont", LabelFont
End Sub
```

127

Because the Font object is a standard object, it can be saved and retrieved using PropertyBags.

The developer using your control can now use the Property window to set a font for the LabelFont property. Supposing that the name you give your control is MultiEdit, she can also use code like the following at run time:

```
Private Sub Command1_Click()
    YourControl1.LabelFont.Bold = True
    YourControl1.LabelFont.Name = "Monotype Sorts"
End Sub
```

128

The beauty of this code is that it never calls the Property Let for the LabelFont property, as you can verify by adding the code above to a UserControl that has several constituent Label controls, and putting breakpoints in the Property Get and Property Let.

When Visual Basic executes the first line above, the Property Get is called, returning a reference to the UserControl's Font object. The Bold property of the Font object is set, using this reference. Because the constituent Label controls all have references to the UserControl's Font object, the change is reflected immediately.

Don't Expose Constituent Controls as Properties

You might wonder why you shouldn't simply expose constituent controls whole. For example, if your UserControl has a text box control named Text1 on it, you might try to write the following:

```
' Kids, don't try this at home.
Property Get Text1() As TextBox
    Set Text1 = Text1
End Property
```

129

The user of your control could then access all the properties and methods of Text1, and you've only written one line of code.

The code above will not compile, because TextBox is not a public data type. But that's not the real reason this is a bad idea.

It might simplify your life to expose all the properties and methods of a constituent control, rather than selectively exposing them, but consider the experience that awaits the user of your control. He now has direct access to the Text property of Text1, bypassing any validation code you might have written in a Property Let. He can also change the height and width of the text box, which may completely wreck the code you've written in your UserControl_Resize event procedure.

All in all, the developer is likely to conclude that your control is too buggy to be worth using. But wait, there's more. If the developer uses your control with other development tools, such as Microsoft Excel or Microsoft Access, type library information for the constituent controls will not be available. All references to Text1 will be late bound, so your control will be not only buggy, but slow.

Exposing constituent controls also limits your ability to change your control's implementation in the future. For example, you might want to base a future version of your control on a different constituent control than the intrinsic TextBox control. Unless the properties and methods of this SuperTextBox exactly matched those of the intrinsic TextBox, your users would be unable to upgrade without rewriting their code.

It's good programming practice to expose only those constituent control properties required for the operation of your control. For example, if the text box mentioned above holds a user name, you might expose the value of Text1.Text through a UserName property.

Using the ActiveX Control Interface Wizard

When you have a large number of constituent controls, or even one constituent control with many properties you wish to expose, the ActiveX Control Interface Wizard can significantly reduce the amount of work required to expose constituent control properties.

Using the ActiveX Control Interface Wizard is discussed in the related topic "Properties You Should Provide."

Using Standard Control Property Types

The code examples in the related topic “Exposing Properties of Constituent Controls” show the use of the standard types Font and OLE_COLOR to create properties that use standard, system-supplied property pages, which the user can access with the ellipsis button in the Properties window.

Whenever possible, use the standard data types and enumerations provided by Visual Basic as data types of your control’s properties. This makes life easy for your users, by presenting consistent property value choices in the Properties window.

Standard Enumerations

The following code uses the standard enumeration for the MousePointer property.

```
Public Property Get MousePointer() As _
        MousePointerConstants
    MousePointer = UserControl.MousePointer
End Property

Public Property Let MousePointer(ByVal NewPointer As _
        MousePointerConstants)
    UserControl.MousePointer = NewPointer
    PropertyChanged "MousePointer"
End Property
```

130

When the MousePointer property appears in the Properties window, it will have the same enumeration as the MousePointer properties of other controls.

You can use the Object Browser to determine what enumerations are available in the Visual Basic type libraries.

Note The purpose and importance of PropertyChanged are discussed in “Adding Properties to Controls,” earlier in this chapter.

131

Standard Data Types

Visual Basic provides four standard data types of special interest to control authors.

OLE_COLOR

The OLE_COLOR data type is used for properties that return colors. When a property is declared as OLE_COLOR, the Properties window will display a color-picker dialog that allows the user to select the color for the property visually, rather than having to remember the numeric equivalent.

An example of the use of OLE_COLOR can be found in “Exposing Properties of Constituent Controls.”

OLE_COLOR is treated internally as a Long.

OLE_TRISTATE

The OLE_TRISTATE data type is used for three-state check boxes. If you're authoring a control with check box functionality, declare its Value property as OLE_TRISTATE.

OLE_TRISTATE is an enumeration with the following values:

- 0 - Unchecked
- 1 - Checked
- 2 - Gray

34

OLE_OPTEXCLUSIVE

If you're developing a control with option-button functionality, use the OLE_OPTEXCLUSIVE data type for the Value property of your control. This will cause your control's Value property to behave like that of the intrinsic OptionButton control. That is, when instances of your control are grouped, and the user clicks an unselected control instance, the currently selected instance's Value is automatically set to 0 (thus unselecting the button), and the Value of the clicked instance is set to 1.

This behavior is handled by the container. The container checks the Value property for each control it contains, and groups those that are of type OLE_OPTEXCLUSIVE.

Note You must use the Procedure Attributes dialog box to make the Value property the default property, in order for the control host to enable the behavior described.

132

OLE_OPTEXCLUSIVE is handled as a Boolean type internally.

OLE_CANCELBOOL

Use this data type for an event argument that allows the user to cancel the event. For example, the standard KeyPress event passes a Cancel parameter as its last parameter. If the user sets this parameter to False, the event is canceled.

OLE_CANCELBOOL is handled as a Boolean type internally.

Creating Design-Time-Only, Run-Time-Only, or Read-Only Run-Time Properties

To create a property that can be read at run time, but can be set only at design time, implement the property using property procedures. In the Property Let or Property Set procedure, test the UserMode property of the Ambient object. If UserMode is True, raise an error, as shown in the following code fragment:

```
Private mdblSerendipity As Double

Property Get Serendipity() As Double
    Serendipity = mdblSerendipity
End Property

Property Let Serendipity() As Double
```

—70

```
    ' (Code to validate property values omitted.)
    If Ambient.UserMode Then Err.Raise Number:=31013, _
        Description:= _
        "Property is read-only at run time."
    Serendipity = mdblSerendipity
    PropertyChanged "Serendipity"
End Property
```

133

To suppress a property completely at run time, you can also raise a "Property is not available at run time" error in Property Get.

Note Implementing properties of the Variant data type requires all three property procedures, Property Get, Property Let, and Property Set, because the user can assign any data type, including object references, to the property. In that case, the error raised in Property Let must also be raised in Property Set.

134

Handling Read-Only Run-Time Properties in the ReadProperties Event

The recommended practice for the ReadProperties event is to assign the retrieved value to the property, so that the Property Let is invoked. This allows the validation code in the Property Let to handle invalid values the user has manually entered into the container's source file, as described in "Saving the Properties of Your Control."

Clearly, this is problematic for read-only run-time properties. The solution is to bypass the Property Let, and assign the retrieved value directly to the private member or constituent control property. If the property accepts only certain values, you can use a helper function that can be called from both Property Let and ReadProperties.

The following code fragment illustrates these two solutions:

```
Private Sub UserControl_ReadProperties(PropBag As _
        PropertyBag)
    On Error Resume Next
    ' Retrieve the value of the HasWidgets property,
    ' which is read-only at run time.
    mblnHasWidgets = _
        PropBag.ReadProperty("HasWidgets", False)

    ' Retrieve the value of the Appearance property,
    ' which can be set at design time, is read-only at
    ' run time, and has two valid values, Appears3D
    ' and AppearsFlat.
    mintAppearance = ValidateAppearance( _
        PropBag.ReadProperty("Appearance", Appears3D))

    ' . . . more properties . . .
End Sub
```

135

For properties with Boolean, String, or general-purpose numeric values, you can simply assign the value to the private member, as with the HasWidgets property in the example above.

The Property Let for the Appearance property would call the same ValidateAppearance helper function used in the example above. The helper function might look something like this:

```
Private Sub ValidateAppearance(ByVal Test As Integer)
    Select Case Test
        Case Appears3D
        Case AppearsFlat
        Case Else
            Err.Raise 30013, , "Invalid value"
    End Select
End Sub
```

136

Important The discussion of valid property values above assumes that the correct data type is used. If the wrong data type is entered in the source file, a type mismatch error will occur. An error will also occur if the Appearance property value is invalid. (This is why you should *always* use error trapping in ReadProperties.) You can ignore the error with On Error Resume Next, as above, and the property will have whatever value the private member contained at startup. If this would not be a valid value for your control, you must take some action in response to the error.

137

Creating Run-Time-Only Properties

You can create a property that is available only at run time by causing property procedures to fail during design time (that is, when the UserMode property of the Ambient object is False).

Visual Basic's Properties window does not display properties that fail during design-time.

Tip You can open the Procedure Attributes dialog box, select your run-time-only property, click the Advanced button, and check "Don't show in Property Browser" to prevent the Properties window from interrogating the property. This keeps the Properties window from putting you in break mode every time it queries the property, which is a nuisance when you're debugging design-time behavior.

138

Marking a Property as the Properties Window Default

When you first place a new control instance on a form or other container, Visual Basic chooses a property to highlight the Properties window. This will be the same as the last property that was highlighted, if the new control has that property. Otherwise, Visual Basic uses the property marked as the *user interface default.*

If you don't specify this user interface default, Visual Basic highlights a property according to various internal criteria, such as the order in which you added properties to the type library.

To specify the user interface default for your control

11 In the UserControl code window, place the cursor in one of the property procedures for the property.

12 On the **Tools** menu, click **Procedure Attributes** to open the **Procedure Attributes** dialog box.

13 The property should be shown in the **Name** box. If not, use the **Name** box to select it.

14 Click the **Advanced** button to show advanced features. Check **User Interface Default** in the **Attributes** box, then click **OK** or **Apply**.

35

Tip The best candidate for the user interface default is the property users will most often set. For example, the Interval property is the user interface default for the Timer control.

139

Grouping Properties by Category

The Properties window has two tabs, one showing all of a control's properties in alphabetical order, and one which organizes the properties into categories.

You can assign each of your control's properties to one of the existing categories, or create new categories. Assigning categories is highly recommended, because Visual Basic places all unassigned properties in the Misc category.

To assign a property to a property category

15 In the UserControl code window, place the cursor in one of the property procedures for the property you want to assign to a category.

16 On the **Tools** menu, click **Procedure Attributes** to open the **Procedure Attributes** dialog box.

17 The property should be shown in the **Name** box. If not, use the **Name** box to select it.

18 Click the **Advanced** button to show advanced features. Select the desired category in the **Property Category** box, then click **OK** or **Apply**.

36

You can create a new category by typing a category name in the Property Category box. The category will be created only for this control. You can use the same category name for other controls, but it must be entered separately for each control.

Tip To reduce user confusion, assign properties to the same categories they appear in for other controls. If possible, use existing categories. Create new categories only when you have a group of related properties that will clearly be easier to use if grouped in a new category.

140

Properties You Should Provide

Recommended properties include Appearance, BackColor, BackStyle, BorderStyle, Enabled, Font, and ForeColor. It's also a good idea to implement properties commonly found on controls that provide functionality similar to yours.

In addition, you may wish to selectively implement properties of any constituent controls on your UserControl object, as discussed in "Exposing Properties of Constituent Controls," earlier in this chapter.

All of the above properties should use the appropriate data types or enumerations, as discussed in the related topics "Using Standard Control Property Types" and "Exposing Properties of Constituent Controls."

Note If you're authoring a control that provides its appearance using constituent controls, implementing the Appearance property is problematic. For most controls, the Appearance property is available only at design time — but you can only delegate to run-time properties of constituent controls.

141

Procedure IDs for Standard Properties

Every property or method in your type library has an identification number, called a procedure ID or DISPID. The property or method can be accessed either by name (late binding) or by DISPID (early binding).

Some properties and methods are important enough to have special DISPIDs, defined by the ActiveX specification. These standard procedure IDs are used by some programs and system functions to access standard properties of your control.

For example, there's a procedure ID for the method that displays an About Box for a control. Rather than rummaging through your type library for a method named AboutBox, Visual Basic calls this procedure ID. Your method can have any name at all, as long as it has the right procedure ID.

To assign a standard procedure ID to a property

19 On the **Tools** menu, click **Procedure Attributes** to open the **Procedure Attributes** dialog box.

20 In the **Name** box, select the property.

21 Click **Advanced** to expand the **Procedure Attributes** dialog box.

22 In the **Procedure ID** box, select the procedure ID you want to assign to the property. If the procedure ID you need is not in the list, enter the number in the **Procedure ID** box.

142

Important Selecting (None) in the procedure ID box does not mean that the property or method will not have a procedure ID. It only means that you have not selected a particular procedure ID. Visual Basic assigns procedure IDs automatically to members marked (None).

143

One Procedure ID to a Customer

A property or method of a control can have only one procedure ID, and no other property or method of the control can have the same procedure ID.

That is, every control in your control component can have a default property (procedure ID = 0), but only one property on each control can have that procedure ID.

If you assign the same procedure ID to two different members, the one that comes first in the type library is the only one that can be accessed. The other might as well not exist.

Procedure IDs of Interest

It's always a good idea to assign the standard procedure ID to a property, if there is one. The Procedure Attributes dialog box lists procedure IDs by the property name they are usually associated with. You may find the following IDs of particular interest.

AboutBox

Allows you to specify a method that shows an About Box for your control, as discussed in "Adding an About Box to Your Control," earlier in this chapter. There is no particular method name associated with this ID.

Caption, Text

Either of these procedure IDs will give a property the Properties window behavior demonstrated by the Caption and Text properties of Visual Basic intrinsic controls. That is, when a user types a value into the Properties window, the new value will be reflected immediately in the control.

This means that your Property Let procedure will be called for each keystroke the user enters, receiving a complete new value each time.

The property you assign these to need not be called Caption or Text, although those properties represent the kind of functionality these procedure IDs were designed to support.

(Default)

The default property of a control is the one that will be accessed when no property has been specified. For example, the following assigns the string "Struthiomimus" to the (default) Caption property of Label1:

```
Label1 = "Struthiomimus"
```

144

Enabled

This procedure ID must be assigned to the Enabled property of your control, in order for its enabled/disabled behavior to match that of other controls. This is discussed in "Allowing Your Control to be Enabled or Disabled," earlier in this chapter.

For More Information The procedure IDs of interest to control authors are listed in the Procedure ID box of the Procedure Attributes dialog box. For a complete list of DISPIDs, consult the ActiveX specification.

145

Providing Useful Defaults

Whenever you implement a property with the same name as one of the standard ambient properties, such as BackColor, Font, and ForeColor, you should consider whether the value of the corresponding property of the Ambient object would be a useful default.

You can see an example of this behavior by changing the size of the font on a Visual Basic form, and then adding a label or command button. The new control uses the form's current font settings as its default font settings. Most of the intrinsic controls follow this example.

If you're authoring a check box, option button, or label, setting the control's default BackColor to match Ambient.BackColor might be a useful service to your users.

Clearly, this requires some thought about how controls are used. For example, on a text box the Font property would be a good candidate for ambient matching, while the BackColor property would not.

For More Information See "Using the Ambient Object to Stay Consistent with the Container," earlier in this chapter.

146

Using the ActiveX Control Interface Wizard

The ActiveX Control Interface Wizard can assist you in determining what properties to provide, and in delegating to the appropriate constituent controls.

After you have placed all the constituent controls you're going to use on your UserControl, start the wizard and select your control. The wizard will examine your constituent controls, and produce a list of all the properties, methods, and events that appear in all their interfaces, plus those in the UserControl object's interface, and the standard properties listed above. You can select from this list those properties, methods, and events you want in your control's interface.

The wizard will produce default mappings of your control's properties to properties of the UserControl object or of constituent controls. In subsequent steps, you can modify these mappings.

When you have finished determining your control's interface and delegating to existing properties, the wizard will generate property procedure code to implement the properties, using the correct data types for standard properties, and including delegation code for all your property mappings, enormously reducing the amount of work required to generate a full-featured control.

Adding Methods to Controls

You implement methods of your ActiveX control by adding Public Sub and Function procedures to the code module of the UserControl that forms the basis of your control class.

—76

By default, the only methods your control will have are the extender methods provided by the container, such as the Move method. You can decide what additional methods your control needs, and add code to implement them.

Standard Methods

If your control is not invisible at run time, you should provide a Refresh method. This method should simply call UserControl.Refresh. For user-drawn controls, this will raise the Paint event; for controls built using constituent controls, it will force a refresh of the constituent controls.

It's also a good idea to implement methods commonly found on controls that provide functionality similar to yours. In addition, you may wish to selectively implement methods of the UserControl object, or of its constituent controls.

Using the ActiveX Control Interface Wizard

The ActiveX Control Interface Wizard can assist you in determining what methods to provide, and in delegating to the appropriate constituent controls.

After you have placed all the constituent controls you're going to use on your UserControl, start the wizard and select your control. The wizard will examine your constituent controls, and produce a list of all the properties, methods, and events that appear in all their interfaces, plus those in the UserControl object's interface. You can select from this list those properties, methods, and events you want in your control's interface.

The wizard will produce default mappings of your control's methods to methods of the UserControl object or of constituent controls. In subsequent steps, you can modify these mappings.

When you have finished with determining your control's interface, and delegating to existing methods, the wizard will generate Sub and Function procedures to implement the properties, including delegation code for all your mappings. This greatly reduces the amount of work required to generate a full-featured control.

Raising Events from Controls

It's important to distinguish between the events received by your UserControl object (or by the controls it contains) and the events your control raises. Events your control *receives* are opportunities for you to do something interesting; events your control *raises* provide opportunities for the developer who uses your control to do something interesting.

This principle is demonstrated, with illustrations, in the step by step procedure "Adding an Event to the ShapeLabel Control" in "Creating an ActiveX Control."

Exposing Events of Constituent Controls

The mechanism for exposing events is different from the delegation used to expose properties and methods. You expose an event in a constituent control by raising your own event, as in the following code fragment from a UserControl code module:

```
' Declaration of your control's Click event.
Public Event Click()

' When the txtBase text box control raises a Click,
'   your control forwards it by raising the Click
'   event you declared.
Private Sub txtBase_Click()
    RaiseEvent Click
End Sub

' You may also want to raise your control's Click event
'   when a user clicks on the UserControl object.
Private Sub UserControl_Click()
    RaiseEvent Click
End Sub
```

147

Notice that your Click event may be raised from multiple locations in your code. You can add your own code before and after raising the Click event.

The Difference Between Events and Properties or Methods

It may help to think of properties and methods as *inbound,* and events as *outbound.* That is, methods are invoked from outside your control, by the developer who's using your control. Thus, the developer invokes a method of your UserControl object, and you respond by delegating to the method of your constituent control.

By contrast, events originate in your control and are propagated outward to the developer, so that she can execute code in her event procedures. Thus, your UserControl object responds to a click event from one of its constituent controls by raising its own Click event, thus forwarding the event outward to the developer.

Using the ActiveX Control Interface Wizard

The ActiveX Control Interface Wizard can greatly simplify the task of forwarding events. This is discussed in the related topic "Events Your Control Should Raise."

Events the Container Provides for Your Control

The container's extender object may provide events for the benefit of developers using your control. If your control is used with Visual Basic, the user of your control gets four such events without any work on your part: GotFocus, LostFocus, DragOver, and DragDrop.

You don't need to be concerned with extender events. They are invisible to your control, and you cannot put code in their event procedures.

Specifying a Default Event for the Code Window

—78

The first time you select a control instance in the Object box of the container's code window, Visual Basic selects an event to highlight in the Procedure box, and inserts into the code window an event procedure for that event.

If a control doesn't specify this *user interface default,* Visual Basic selects the first event alphabetically.

Note While you're working with a new control in the development environment, before you've specified the user interface default, Visual Basic may sometimes select the default event based on internal considerations, such as which event went into the type library first.

148

To specify the user interface default for your control's events

23 In the UserControl code window, place the cursor on the declaration of the event you want to specify as the user interface default.

24 On the **Tools** menu, click **Procedure Attributes** to open the **Procedure Attributes** dialog box.

25 The property should be shown in the **Name** box. If not, use the **Name** box to select it.

26 Click the **Advanced** button to show advanced features. Check **User Interface Default** in the **Attributes** box, then click **OK** or **Apply**.

37

Tip You can only mark one event as the user interface default. Choose the event users will most frequently place code in.

149

Events Your Control Should Raise

Recommended events include Click, DblClick, KeyDown, KeyPress, KeyUp, MouseDown, MouseMove, and MouseUp. It's also a good idea to implement events commonly found on controls that provide functionality similar to yours.

In addition, you may wish to selectively implement events of the constituent controls on your UserControl object, or of the UserControl object itself.

It's important to use the same arguments, with the same data types, as these standard events, as discussed in "Exposing Events of Constituent Controls." Data types are discussed in "Using Standard Control Property Types."

Using the ActiveX Control Interface Wizard

The ActiveX Control Interface Wizard can assist you in determining what events to provide, and in forwarding the appropriate constituent control events.

After you have placed all the constituent controls you're going to use on your UserControl, start the wizard and select your control. The wizard will examine your constituent controls, and produce a list of all the properties, methods, and events that appear in all their interfaces, plus those in the UserControl object's interface, and the

standard events listed above. You can select from this list those properties, methods, and events you want in your control's interface.

The wizard will produce default mappings of your control's events to events of the UserControl object or of constituent controls. In subsequent steps, you can modify these mappings.

When you have finished determining your control's interface, the wizard will generate code to raise the events you've selected, using the correct arguments and data types for standard events, and including event forwarding code for all your event mappings. This enormously reduces the amount of work required to generate a full-featured control.

Providing Named Constants for Your Control

As with other component types, public enumerations can be shared by all of the controls in a control component (.ocx file). Place public Enums for your component in any UserControl code module.

"Providing Named Constants for Your Component," in "General Principles of Component Design," discusses techniques for providing constants, validating constants in properties, and so forth. See that topic for general information on the subject.

There are two additional factors specific to control components:

- Enum member names are used in the Properties window.
- Global objects cannot be used to simulate string constants.

38

Enum Member Names in the Properties Window

As an example of the first factor, consider the following Enum and property:

```
Public Enum DINOSAUR
    dnoTyrannosaur
    dnoVelociraptor
    dnoTriceratops
End Enum

Private mdnoFavoriteDinosaur As DINOSAUR

Public Property Get FavoriteDinosaur() As DINOSAUR
    FavoriteDinosaur = mdnoFavoriteDinosaur
End Property

Public Property Let FavoriteDinosaur(ByVal NewDino _
        As DINOSAUR
    mdnoFavoriteDinosaur = NewDino
```

—80

```
    PropertyChanged "FavoriteDinosaur"
End Property
```

150

When you set the FavoriteDinosaur property in the Properties window, the drop down list will contain dnoTyrannosaur, dnoVelociraptor, and dnoTriceratops.

As you can see, there's a fine tradeoff here between names that will look good in the drop down, and names that will avoid collisions with names used in Enums for other components.

As a rule of thumb, don't abandon the prefix ("dno" in the example above) that groups constants in global lists. The prefix provides at least some protection from name conflicts. On the other hand, don't make your prefixes so long that they obscure the names.

Cannot Simulate String Constants Using Global Objects

Class modules in control components can have one of two values of the Instancing property, Private or PublicNotCreateable. The Instancing values that enable global objects are not available in control components, so it is not possible to simulate string constants using properties of global objects, as described in "Providing Named Constants for Your Component" in "General Principles of Component Design."

Setting Up a New Control Project and Test Project

As discussed in "Two Ways to Package ActiveX Controls," earlier in this chapter, Visual Basic enables you to author shareable control components (.ocx files), or to include private controls as .ctl files in your component project. These two scenarios have different testing requirements.

Testing Private Controls

The only way to test a private control is to place it on a form within the project. Of course, there may be several forms in the project that use the control, but it is recommended that you create a separate form for testing your private controls.

The reason for this is that simply using the control is not likely to test it exhaustively. Once your application or component is compiled, user actions you have not anticipated may cause unexpected results in your control's code.

By including a test form that exercises all of your control's interface members, you can test your control more thoroughly.

Testing Controls in Control Components

When you're developing a control component, you need thorough test coverage of all aspects of your control. This coverage is best provided using a separate test project.

Visual Basic allows you to run multiple projects, so you can load your test project and ActiveX control project and run them together for debugging purposes.

Once you have compiled your control component, the test project can be used as a test harness for quality assurance test suites.

If you use the ActiveX Control Interface Wizard to build the interface and generate code for your control, you can get a test project created by simply checking an option on the wizard's final screen.

Examples of creating a new ActiveX control project and a test project can be found in the step by step procedures "Creating the ControlDemo Project" and "Adding the TestCtlDemo Project," in "Creating an ActiveX Control."

Tip You may prefer to author your controls as private controls in a Standard EXE project, and to test them by placing them on forms within the project. When you're ready to compile an .ocx file, you can remove the .ctl files from the Standard EXE project and add them to an ActiveX control project.

You can then set up the Standard EXE project as a test harness, using the Controls tab of the Components dialog box to add your controls to the Toolbox, as described in "Compiling the ControlDemo Component," in "Creating an ActiveX Control."

151

For More Information "Two Ways to Package ActiveX Controls" lists several reasons why you might want to create a control component, even if you're just distributing controls as part of your own applications.

152

Creating Robust Controls

For your user, the three most important things about an ActiveX control are robustness, robustness, and robustness. Because a control component runs in the process space of an application that uses it, fatal errors in your controls are also fatal errors for the application.

The following lists of DOs and DON'Ts are by no means inclusive. They only provide a starting point for producing robust controls.

Error Handling

DO	DON'T
Provide thorough error handling in every event procedure you put code in, whether the event belongs to the UserControl or to a constituent control. In particular, *provide thorough error handling in the UserControl's Paint, Resize, and Terminate events.*	*Raise errors in any event procedures.*

153

Unhandled errors in event procedures will be fatal to your control component, and the application using it, because there will never be a procedure on the call stack that can handle the errors.

It's perfectly safe to raise errors in property procedures and methods, because properties and methods are always invoked by other procedures, and errors you raise can be handled by the user in those procedures.

Object Models

If your control component includes dependent objects, such as a collection of ToolbarButton objects for a Toolbar control:

DO	DON'T
Create wrapper classes for collections of such objects, as described in "General Principles of Component Design," and in "Standards and Guidelines."	*Use the Collection object without a wrapper class.* The Collection object accepts any variant, meaning your users could accidentally insert objects that might cause errors in your control's code.
Use property procedures for collection properties.	*Implement such properties as simple data members.*

154

For example, if you create a ToolbarButtons class as the wrapper class for a collection of ToolbarButton objects, add the property to your UserControl object as a read-only property procedure:

```
Private mToolbarButtons As ToolbarButtons

Property Get ToolbarButtons() As ToolbarButtons
    Set ToolbarButtons = mToolbarButtons
End Property

Private Sub UserControl_Initialize()
    Set mToolbarButtons = New ToolbarButtons
End Sub
```

155

By contrast, the following implementation allows your user to accidentally set ToolbarButtons to Nothing, destroying the collection:

```
Public ToolbarButtons As New ToolbarButtons
```

156

Implementing Properties

DO implement properties using property procedures, instead of public data members.

You can use Property Let to validate property values. If you use public data members, you'll have to check the data every time you use it; and if that happens in an event procedure, you won't be able to raise an error without bringing down the application that's using your control component.

In addition, your properties will not work correctly in the Properties window and Property Pages dialog box, as discussed in "Adding Properties to Your Control," earlier in this chapter.

Debugging Controls

The most important difference between debugging controls and debugging other objects is that some of the code in your control must execute while the form a control instance is placed on is in design mode.

For example, the code in the property procedures you use to implement your control's properties must execute when the developer using your control sets its properties using the Properties window.

Code that saves and retrieves your control's property values must also run at design time, whenever the user loads a form containing an instance of the control, puts the project in Run mode, or saves the project to disk.

Code in the Resize event (or the Paint event, for user-drawn controls) must run at design time to provide the design-time appearance of your control.

You can see this feature in action in the step by step procedure "Running the ShapeLabel Control at Design Time," in "Creating an ActiveX Control."

For More Information General information on debugging components can be found in "Debugging, Testing, and Deploying Components."

157

Running Code at Design Time

To put a control you're authoring into a state such that its code can execute at design time, you must close the control's visual designer, by clicking the Close box or pressing CTRL+F4.

When the designer is closed, Visual Basic enables the control's icon in the Toolbox, so that you can add instances of the control to forms for testing.

If code in your control hits a break point at design time, for example during a Property Let invoked by the Properties window, Visual Basic enters break mode, just as it would if your project were running. When you press F5 to continue execution, the code in your control resumes execution. Visual Basic remains in design mode.

You can see this by setting a break point in a property procedure, and then placing an instance of your control on a test form.

Making Changes to Existing Controls

When you open the UserControl designer for a control, Visual Basic disables all instances of the control and grays the control's icon in the Toolbox. If you have a test form open with instances of the control on it, Visual Basic covers the disabled control instances with cross-hatching.

—84

You can also disable control instances by certain changes to code in the control's code window, such as adding a new property or method, or adding code to a previously unused event procedure.

Once a control instance has been disabled in this fashion, it cannot execute code. It will not even receive a Terminate event.

Refreshing Control Instances

When you close the UserControl designer, the disabled instances of the control are quietly destroyed (they get no more events) and replaced with fresh instances. You can see this by putting Debug.Print statements in the Initialize and Terminate events of a control.

You can also refresh the control instances by right-clicking the test form and selecting Update UserControls from the context menu. If there are any control designers open, they will be closed before the control instances are refreshed.

For More Information General information on debugging components can be found in "Debugging, Testing, and Distributing Components."

158

Distributing Controls

As discussed in "Two Ways to Package ActiveX Controls," earlier in this chapter, Visual Basic lets you author shareable control components (.ocx files), or simply include private controls as .ctl files in the project for your application or component.

This topic and its related topics focus on distribution, versioning, and licensing issues for control components. Private controls are compiled directly into an executable or component, and are distributed along with it. Being private, they also have no versioning or licensing issues.

Distributing Control Components

When you distribute a control component, you're providing a tool other developers can use in their applications. Versioning issues address the question of how you update that tool without breaking your customers' code.

Because you're providing a tool, instead of a finished application, you have licensing issues to consider. You have to decide whether to include licensing support for your control. If you plan on building your controls using licensed controls from other authors, you need to consider how that affects your distribution plans.

Because the tool you're creating is an-process component ("ocx" is really just another way to spell DLL), you have to select a base address that will minimize memory conflicts, and thus avoid performance problems.

Finally, because the tool you're creating uses the Visual Basic run-time DLL, and possibly other support files, you have to create a Setup program.

—85

Setup is covered in the remainder of this topic. The important subject of base addresses is discussed in “Setting Base Addresses for In-Process Components” in “Debugging, Testing, and Deploying Components.”

Creating Setup for ActiveX Control Components

ActiveX controls created with Visual Basic require the Visual Basic run-time DLL. Depending on what constituent controls you use, you may require additional support files. To ensure that you distribute all the necessary support files, using SetupWizard is recommended.

For the most part, using SetupWizard for control components is no different from using it for any other component created using Visual Basic. This subject is thoroughly covered in “Debugging, Testing, and Deploying Components.”

If you plan to use your control component for Internet or intranet development, you can obtain the most up-to-date information on setup options from the Microsoft Visual Basic Web site. On the Visual Basic Help menu, click Microsoft on the Web, then click Product News.

For More Information The SetupWizard is introduced in “Distributing Your Application.”

159

Licensing Issues for Controls

Licensing for controls is a sensitive issue. After you’ve spent hundreds of hours developing a control, what if somebody else puts an instance of it on a UserControl, exposes all the properties, methods, and events, adds one or two trivial properties, then compiles and sells it as a new control?

Visual Basic’s licensing support protects your investment. When you add licensing support to your control component, a license key is compiled into it. This key covers all the controls in the component.

Running your Setup program transfers the license key to another computer’s registry, allowing your controls to be used for development. Simply copying your .ocx file to another computer and registering it does not transfer the license key, so the controls cannot be used.

To add licensing support to your control project

- On the **Project** menu, click **<MyProject> Properties** to open the **Project Properties** dialog box. Select the **General** tab, check **Require License Key**, then click **OK**.

39

When you make the .ocx file, Visual Basic will create a .vbl file containing the registry key for licensing your control component. When you use the setup wizard to create Setup for your .ocx, the .vbl file is automatically included in the setup procedure.

—86

How Licensing Works

When a developer purchases your control component and runs your Setup program, the license key is added to the registry on her computer.

Thereafter, whenever the developer puts an instance of your control on a form, Visual Basic (or any other developer's tool) tells the control to create itself using the registry key.

If a developer has obtained a copy of your control component, but not the registry key, the control cannot create instances of itself in the development environment.

When a Developer Distributes Applications

When the developer compiles a program that uses one of your controls, the license key for your component is compiled in. When she creates a Setup for the program, your .ocx is included. Users can then purchase the compiled program and run Setup. Your control is installed on each user's machine — but your license key is not added to the registry.

Each time a user runs the program, the Visual Basic run-time DLL asks your control to create a run-time instance of itself, and passes it the key that was compiled into the program. Your control doesn't have to check the registry, because Visual Basic passed it the key.

Thus the user can run a compiled application without having to have the control component's license key in the registry.

Licensing and the User

Suppose the user later obtains a copy of Visual Basic. Noticing that your control component is installed on his computer, he adds your .ocx file to a project.

The first time he tries to put an instance of one of your controls on a form, Visual Basic tells the control to create itself using the registry key. The key is not there, so the control component can't be used in the development environment.

Licensing and General-Purpose User Applications

When desktop applications such as Microsoft Word and Microsoft Excel create control instances on documents or user forms, they tell the control to create an instance of itself using the license key in the registry. This means that a licensed control cannot be used by an end user unless the user has purchased your control component and installed it.

User documents cannot have the license key compiled into them. Suppose the user of a desktop application gives a coworker a copy of your control component along with a document that contains one of your controls. When the document is opened, the control will be asked to create its run-time instance — using the registry key.

In other words, the coworker must also have purchased and installed your control component. Otherwise, when the document is opened, the control instance cannot be created.

Corporate developers who author ActiveX controls for use by end users within their companies may find it more convenient to omit licensing support. This will make it easier for end users to distribute documents containing controls.

Licensing and the Control Author

Now suppose that someone who purchased your control component decides to use one of your controls to author a new control of her own. As with any other program, when she compiles her control component, your license key is compiled in. SetupWizard creates a license key for the new component, but does not add *your* license key to the setup program.

When a developer installs this new code component, its license key is placed in the registry. The developer then runs Visual Basic, and attempts to put an instance of the control on a form.

The control is asked to create itself using the registry key. In turn, it asks its *constituent controls* to create themselves using their registry keys. Your control doesn't find its license key in the registry, so control creation fails.

Distributing Controls That Use Licensed Controls

If the control author wishes to distribute a new control that uses a control you authored, she must inform purchasers that in order to use her control, they must have your control component installed on their computers.

Alternatively, the control author might negotiate with you for the right to distribute your license key along with her own, in the setup program for her control.

In either case, both license keys will be installed on a developer's machine, so the developer can create design-time instances of the second author's controls. When those controls are compiled into an executable program, both license keys are compiled in.

When the program is subsequently installed by a user and run, the second author's control is asked to create itself. Its constituent controls are also asked to create themselves, and passed their license keys. (And so on, if *your* control uses constituent controls with license keys.)

For More Information Licensing and distribution of constituent controls, including those supplied with the Professional Edition of Visual Basic, is discussed in "Controls You Can Use As Constituent Controls," earlier in this chapter.

160

Licensing and the Internet

Licensed controls can be used on World Wide Web pages, in conjunction with browsers that support control licensing. Both the control component and the license

key must be available to be downloaded to the computer of the person accessing a Web page.

The downloaded license key is not added to the registry. Instead, browser asks the control to create a run-time instance of itself, and passes it the downloaded license key.

The owner of the Web server that uses your control must have purchased and installed your control, just as a developer would, in order to supply both control and license.

If the license is not available, control creation will fail, and the browser will receive a standard control creation error. Whether the browser passes this message along to the person accessing the Web page, or simply ignores it, depends on the person that developed the browser.

For More Information See "Controls You Can Use As Constituent Controls," earlier in this chapter.

161

Versioning Issues for Controls

When you create a new version of a control, there are several areas of backward compatibility you must address:

- Your controls interface; that is, its properties, methods, and events.
- UserControl properties that affect control behavior.
- Whether property values are saved and retrieved.
- Procedure attribute settings.

40

Interface Compatibility

You can add new properties, methods, and events without breaking applications compiled using earlier versions of your control component. However, you cannot remove members from the interface, or change the arguments of members.

You can use Visual Basic's Version Compatibility feature to avoid creating incompatible interfaces. On the Component tab of the Project Properties dialog box, click Binary Compatibility in the Version Compatibility box. This enables a text box in which you can enter the path and file name of the previous version of your component.

The default value in this text box is the last location where you built the component. If you are going to continue using that location to build the new version of your control component, it's a good idea to place a copy of your previous version in another location, and then enter that location in the text box.

For More Information Interface compatibility and the use of the Version Compatible Component box are discussed in "Version Compatibility," in "General Principles of Component Design."

162

UserControl Properties

Be careful when changing properties of the UserControl object, such as ControlContainer. If a previous version of your control had this property set to True, so that developers could use the control to contain other controls, and you change the property to False in a subsequent version, existing applications may no longer work correctly if the new version is installed on the same computer.

Saving and Retrieving Property Values

You may retain a property for backward compatibility, but stop mentioning it in your Help file, and mark it as Hidden using the Procedure Attributes dialog box.

You can stop saving the value of such obsolete properties in the WriteProperties event, but you should continue to load their values in the ReadProperties event. If you stop loading a property value, you will break any previously compiled application that used the property.

Procedure Attribute Settings

Changing attributes of a procedure may break applications that were compiled using previous versions of your control component. For example, it you use the Procedure Attributes dialog to change the default property or method of a control, code that relied on the default will no longer work.

Localizing Controls

You can increase the market for your control component by *localizing* it. A localized control displays text — captions, titles, and error messages — using the language of the locale in which the control is used for application development, rather than the language of the locale in which it was authored.

This topic examines localization issues specific to ActiveX controls. General localization issues are discussed in “International Issues.”

Using the LocaleID

When you compile an executable with Visual Basic, the LocaleID (also referred to as the LCID) of the Visual Basic version is compiled in. Thus an application compiled with the German version of Visual Basic will contain &H0407, the LocaleID for Germany.

In the same way, the LocaleID is compiled into an ActiveX control component created with Visual Basic. This becomes the default LocaleID for your controls. If this were the end of the story, you would have to compile a new version of your component, using the correct version of Visual Basic, for every locale in which you wanted to distribute it.

Fortunately, control components are more flexible than compiled applications. Your component can be used with versions of Visual Basic for any locale, and even with

development tools that support other locales, because they can determine the correct LocaleID at run time.

Discovering the LocaleID

The LocaleID property of the Ambient object returns the LocaleID of the program your control was used in. The Ambient object is available as a property of the UserControl object, as described in "Using the Ambient Object to Stay Consistent with the Container," earlier in this chapter.

You can test the Ambient property as soon as an instance of your control is sited on the container; that is, in the InitProperties or ReadProperties events. Once you know the LocaleID, you can call code to load locale-specific captions, error message text, and so forth from a resource file or satellite DLL, as described later in this topic.

You need to call this code in both events, because the InitProperties event occurs only when a control instance is first placed on a container. Thereafter the control instance receives the ReadProperties event instead, as discussed in "Understanding UserControl Lifetime and Key Events," earlier in this chapter.

You should also call your locale code in the AmbientChanged event, because your control could be used in an application that resets its locale according to Windows Control Panel settings, which can be changed by the user at any time. Your control could also receive AmbientChanged events if it's used as a constituent control, as described later in this topic.

Avoid Accessing Constituent Controls in the Initialize Event

The constituent controls on your UserControl discover the LocaleID by checking the Ambient object which the UserControl, like any good container, makes available to them. This happens automatically, with no effort on your part.

When the Initialize event occurs, your control has been created, and all the constituent controls have been created and sited on your control's UserControl object. However, your control has not yet been sited on the container, so the UserControl cannot supply the correct LocaleID to the constituent controls.

If code in the Intialize event accesses the properties and methods of the constituent controls, their responses will reflect the LocaleID of the version of Visual Basic you used to compile your component, rather than the LocaleID of the application in which your control is compiled. For example, a method call might return a string in the wrong language.

To avoid this, you should not access constituent controls in the Initialize event.

Responding to the AmbientChanged Event

The AmbientChanged event occurs whenever an Ambient property changes on the container your control has been placed on, as discussed in "Using the Ambient Object to Stay Consistent with the Container," earlier in this chapter.

Applications compiled with Visual Basic use the LocaleID of the version of Visual Basic that compiled them. However, your control could be used in an application written using a development tool such as Microsoft Visual C++, in which it is possible to change an application's LocaleID in response to system messages.

For example, if a user opens the Control Panel and changes the locale, an application would receive a notification of the change, and reset itself accordingly. Your controls can handle this situation by including code to change locale dynamically, as in the following example.

```
Private Sub UserControl_AmbientChanged( _
        PropertyName As String)
    Select Case PropertyName
        Case "LocaleID"
            ' Code to load localized captions,
            ' messages, and so forth from a resource
            ' file or satellite DLL, as described below.

        ' Case statements for other properties.

    End Select
End Sub
```

163

A change in the locale can also occur if you use your control as a constituent of another control. As described above, constituent controls don't get the correct LocaleID when they're first sited on a UserControl object. When the outermost control has been sited on the application's form, all the constituent controls will receive AmbientChanged events with the correct LocaleID.

Base Language and Satellite DLLs

The most flexible localization technique is to compile your control component with the default text strings and error messages in the language of the locale you expect to be your largest market. Place text strings and error messages for other locales in satellite ActiveX DLLs, one for each locale.

This scheme makes your component very attractive to developers who create versions of their programs for multiple languages, because they can work with multiple locales on one development machine.

Satellite DLLs are also attractive to users in multilingual countries. Such users may have programs compiled by programmers in different locales; if two such programs use your control component, satellite DLLs allow both programs to coexist on a user's computer.

Important Your control should not raise an error if the requested satellite DLL is not found, as this could cause the entire application to fail. In the event the satellite DLL is not available, simply use the default locale in which your control component was built.

164

Naming Satellite DLLs

If you use an open-ended naming convention for your satellite DLLs, you can supply additional DLLs later without recompiling your program. An example of such a naming convention would be to include the LocaleID in the DLL names. Using this convention, your satellite DLLs for the Belgian French, German, and US English locales might be named MyControls20C.dll, MyControls407.dll and MyControls409.dll.

If you use Windows API calls to load and extract resources from your satellite DLLs, you can create the name of the DLL to be loaded by converting the LocaleID to a string, and appending it to a base name. (Note that the examples above use the hexadecimal representation of the LocaleID.)

You build your satellite DLLs as Visual Basic ActiveX DLL projects. To do this, create a class module with methods for retrieving resources. Give the class a name such as Localizer. Add this class module to each DLL project.

Use your open-ended naming scheme for the Project Name, so that each DLL has a unique programmatic ID, or ProgID, in the Windows registry. Each time you compile a new satellite DLL, you create a new Localizer class, whose full programmatic ID includes the Project Name of the DLL.

In your ActiveX control project, you can then create an instance of the appropriate Localizer class using code such as the following:

```
Dim strProgID As String
Dim objLoc As Object
' Generate the ProgID of the Localizer object
' for the appropriate satellite DLL.
strProgID = "MyControls" & Hex$(Ambient.LocaleID) _
    & ".Localizer"
Set objLoc = CreateObject(strProgID)
If objLoc Is Nothing Then
    ' Satellite DLL not found; use default locale.
Else
    ' Call methods of Localizer object to retrieve
    ' localized string and bitmap resources.
End If
```

165

The code above uses late binding (that is, the variable objLoc is declared As Object). You can get better performance with early binding, by using the Implements feature of Visual Basic. Instead of making the resource retrieval methods members of the Localizer class, you can define them in an abstract class named IResources.

In your Localizer class, use the Implements statement to implement IResources as a second interface. You can call the methods of this interface with early binding, as shown below.

```
' Early-bound variable for IResources interface.
Dim ires As IResources
' Get the IResources interface of the Localizer
' object obtained from the satellite DLL.
Set ires = objLoc
```

```
' Call the methods of your IResources interface
' to retrieve localized resources.
Set cmdOK.Caption = ires.GetString(ID_CMDOK)
```

166

As with the late-bound Localizer object, you can simply add the Localizer class module, with its second interface, to each satellite DLL project. The ability to add the same interface to several different classes is called *polymorphism*.

For More Information The Implements feature is discussed in "Providing Polymorphism by Implementing Interfaces," in "General Principles of Component Design." Accessing satellite DLLs is discussed in "International Issues." Adding resource files to Visual Basic projects is discussed in "More About Programming."

167

Resource Files

An alternative to satellite DLLs is to place text strings and error messages in a resource file, and compile the file into your control component. There are disadvantages to this technique.

- If you use one resource file for each locale, you must compile a separate .ocx file for each locale. To avoid file name conflicts, you can put a locale indicator in the name of each .ocx file, as for example MyControlsDE.ocx for German, or MyControlsFR.ocx for French.
- Unfortunately, you cannot avoid type library name conflicts so easily. A developer can have only one locale version of your control installed at a time. This may be a drawback in multilingual markets.
- Although you can avoid the problem of compiling multiple .ocx files by putting the text and error message strings for all locales into a single resource file, the result will be a much larger .ocx file, and you will have to recompile the component to add support for new locales.

41

Localizing Interfaces

If you localize property, method, and event names, you must compile a separate version of your control for each locale. To allow multiple versions of your control to coexist on one computer, you must use a different name for your control in each version.

As a result of the different interface and control names, multilingual developers will have to rewrite their code for each language version of a program that uses your control. This will make your control component less attractive to such developers.

Microsoft applications, such as Visual Basic, do not localize interface member names.

Localizing Property Pages

Microsoft applications localize property pages, but not property names. If you use this scheme, the caption that shows up on a property page may not match the name of the property in the Properties window.

When localizing captions for properties on property pages, take care to select captions that make it obvious what the property is. Alternatively, you may wish to include the property name in parentheses.

For principles of form layout that simplify localization, see “International Issues.”

Localizing Type Library Information

There is no way to retrieve a browser string from a localized DLL or resource file, so browser strings must be compiled into your type library. To produce localized type libraries, you must use the Procedure Attributes dialog box to change the browser strings for all your properties, methods, and events. You must then re-compile your executable.

Localizing type library information thus limits your ability to localize using satellite DLLs. You may wish to leave your browser strings in the default language of your control.

For More Information See “International Issues.”