Your Visual Basic applications can respond to a variety of mouse events and keyboard events. For example, forms, picture boxes, and image controls can detect the position of the mouse pointer, can determine whether a left or right mouse button is being pressed, and can respond to different combinations of mouse buttons and SHIFT, CTRL, or ALT keys. Using the key events, you can program controls and forms to respond to various key actions or interpret and process ASCII characters.

In addition, Visual Basic applications can support both event-driven drag-and-drop and OLE drag-and-drop features. You can use the Drag method with certain properties and events to enable operations such as dragging and dropping controls. OLE drag and drop gives your applications all the power you need to exchange data throughout the Windows environment — and much of this technology is available to your application without writing code.

You can also use the mouse or keyboard to manage the processing of long background tasks, which allows your users to switch to other applications or interrupt background processing.

Note If you are using the Control Creation Edition of Visual Basic, some of the material covered in this document may not be applicable. With the full editions of Visual Basic you have the ability to create applications, ActiveX documents, and other types of components. Although some of the terminology may relate to application specific objects such as forms, in most cases the underlying concepts also apply to ActiveX control creation.

1

For More Information Other actions and events that involve the mouse or keyboard (the Click and DblClick events, the Focus events, and the Scroll event) are not covered in this chapter. For more information on the Click, DblClick, and Focus events, see the topics "Clicking Buttons to Perform Actions" and "Understanding Focus" in "Forms, Controls, and Menus."

2

Contents

3

Sample Application: Mouse.vbp

Many of the code examples in this chapter are taken from the Mouse.vbp sample application. If you installed the sample applications, you'll find this application in the \Mouse subdirectory of the Visual Basic samples directory (\Vb\Samples\Pguide).

4

Responding to Mouse Events

You can use the MouseDown, MouseUp, and MouseMove events to enable your applications to respond to both the location and the state of the mouse. (This list excludes drag events, which are introduced in "Dragging and Dropping" later in this chapter.) These mouse events are recognized by most controls.

Event	Description
MouseDown	Occurs when the user presses any mouse button.
MouseUp	Occurs when the user releases any mouse button.
MouseMove	Occurs each time the mouse pointer is moved to a new point on the screen.

5

A form can recognize a mouse event when the pointer is over a part of the form where there are no controls. A control can recognize a mouse event when the pointer is over the control.

When the user holds down a mouse button, the object continues to recognize all mouse events until the user releases the button. This is true even when the pointer is moved off the object.

The three mouse events use the following arguments.

Argument	Description
button	A bit-field argument in which the three least-significant bits give the status of the mouse buttons.
Shift	A bit-field argument in which the three least-significant bits give the status of the SHIFT, CTRL, and ALT keys.
x, y	Location of the mouse pointer, using the coordinate system of the object that receives the mouse event.

6

A *bit-field argument* returns information in individual bits, each indicating whether a certain condition is on or off. Using binary notation, the three leftmost bits are referred to as *most-significant* and the three rightmost bits as *least-significant.* Techniques for programming with these arguments are described in "Detecting Mouse Buttons" and "Detecting SHIFT, CTRL, and ALT States" later in this chapter.

The MouseDown Event

MouseDown is the most frequently used of the three mouse events. It can be used to reposition controls on a form at run time or to create graphical effects, for instance. The MouseDown event is triggered when a mouse button is pressed.

Note The mouse events are used to recognize and respond to the various mouse states as separate events and should not be confused with the Click and DblClick events. The Click event recognizes when a mouse button has been pressed and released, but only as a single action — a click. The mouse events also differ from the Click and DblClick events in that they enable you to distinguish between the left, right, and middle mouse buttons and the SHIFT, CTRL, and ALT keys.

7

Using MouseDown with the Move Method

The MouseDown event is combined with the Move method to move a command button to a different location on a form. The new location is determined by the position of the mouse pointer: When the user clicks anywhere on the form (except on the control), the control moves to the cursor location.

A single procedure, Form_MouseDown, performs this action:

```
Private Sub Form_MouseDown (Button As Integer, _
        Shift As Integer, X As Single, Y As Single)
    Command1.Move X, Y
End Sub
```

8

The Move method places the command button control's upper-left corner at the location of the mouse pointer, indicated by the *x* and *y* arguments. You can revise this procedure to place the *center* of the control at the mouse location:

```
Private Sub Form_MouseDown (Button As Integer, _
        Shift As Integer, X As Single, Y As Single)
    Command1.Move (X - Command1.Width / 2), _
    (Y - Command1.Height / 2)
End Sub
```

9

Using MouseDown with the Line Method

The Click-A-Line sample application responds to a mouse click by drawing a line from the previous drawing location to the new position of the mouse pointer. This application uses the MouseDown event and the Line method. Using the following syntax, the Line method will draw a line from the last point drawn to the point (*x2*, *y2*):

Line – (*x2*, *y2*)

10

Click-A-Line uses a blank form with one procedure, Form_MouseDown:

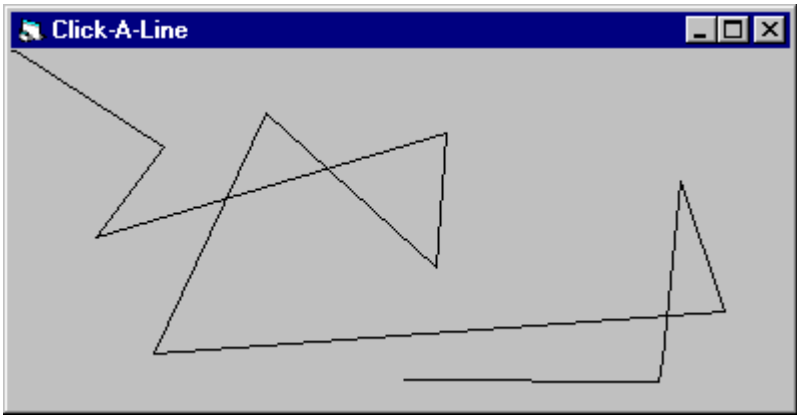
```
Private Sub Form_MouseDown (Button As Integer, _
        Shift As Integer, X As Single, Y As Single)
    Line -(X, Y)
End Sub
```

11

The first line starts at the upper-left corner, which is the default origin. Thereafter, whenever the mouse button is pressed, the application draws a straight line extending

from the previous line to the present location of the mouse pointer. The result is a series of connected lines, as shown in Figure 11.1.

Figure 11.1 Connecting lines are drawn whenever MouseDown is invoked



1

The MouseMove Event

The MouseMove event occurs when the mouse pointer is moved across the screen. Both forms and controls recognize the MouseMove event while the mouse pointer is within their borders.

Using MouseMove with the Line Method

Graphics methods can produce very different effects when used in a MouseMove procedure instead of in a MouseDown procedure. For example, in the topic "The MouseDown Event" earlier in this chapter, the Line method drew connected line segments. In the Scribble application described below, the same method is used in a Form_MouseMove procedure to produce a continuous curved line instead of connected segments.

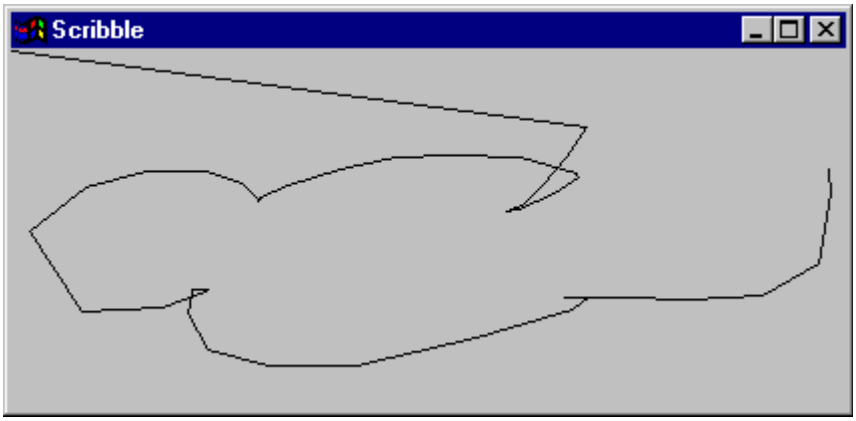
In the Scribble application, the MouseMove event is recognized whenever the mouse pointer changes position. The following code draws a line between the current and previous location.

```
Private Sub Form_MouseMove (Button As Integer, _
        Shift As Integer, X As Single, Y As Single)
    Line -(X, Y)
End Sub
```

12

Like the MouseDown procedure, the line created by the MouseMove procedure starts at the upper-left corner, as shown in Figure 11.2.

Figure 11.2 The MouseMove event and the Line method create a simple sketch program



2

How MouseMove Works

How many times does the MouseMove event get called as the user moves the pointer across the screen? Or, to put it another way, when you move the pointer from the top of the screen to the bottom, how many locations are involved?

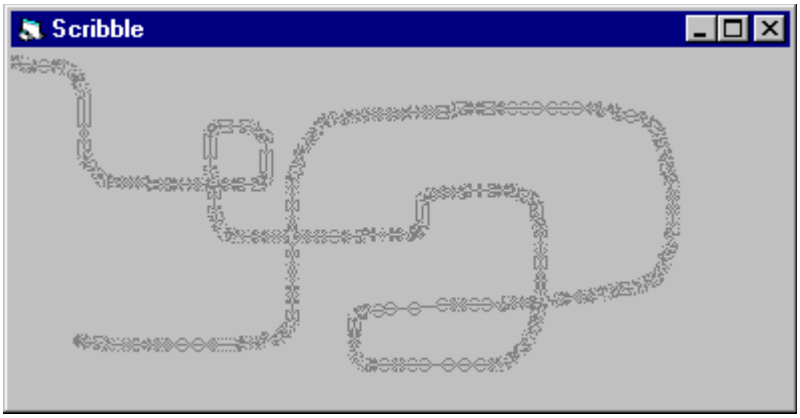
Visual Basic doesn't necessarily generate a MouseMove event for every pixel the mouse moves over. The operating environment generates a limited number of mouse messages per second. To see how often MouseMove events are actually recognized, you can enhance the Scribble application with the following code so that it draws a small circle at each location where a MouseMove event is recognized. The results are shown in Figure 11.3.

```
Private Sub Form_MouseMove (Button As Integer,_
        Shift As Integer, X As Single, Y As Single)
    Line -(X, Y)
    Circle (X, Y), 50
End Sub
```

13

Figure 11.3 A demonstration of where MouseMove events occur

—5



3

Note that the faster the user moves the pointer, the fewer MouseMove events are recognized between any two points. Many circles close together indicate that the user moved the mouse slowly.

Your application can recognize many MouseMove events in quick succession. Therefore, a MouseMove event procedure shouldn't do anything that requires large amounts of computing time.

The MouseUp Event

The MouseUp event occurs when the user releases the mouse button. MouseUp is a useful companion to the MouseDown and MouseMove events. The example below illustrates how all three events can be used together.

The Scribble application is more useful if it allows drawing only while the mouse button is held down and stops drawing when the button is released. To do this, the application would have to respond to three actions:

- The user presses the mouse button (MouseDown).
- The user moves the mouse pointer (MouseMove).
- The user releases the mouse button (MouseUp).

4

MouseDown and MouseUp will tell the application to turn drawing on and off. You specify this by creating a form-level variable that represents the drawing state. Type the following statement in the Declarations section of the form code module:

```
Dim DrawNow As Boolean
```

14

DrawNow will represent two values: True will mean "draw a line," and False will mean "do not draw a line."

Because variables are initialized to 0 (False) by default, the application starts with drawing off. Then the first line in the MouseDown and MouseUp procedures turns drawing on or off by setting the value of the form-level variable DrawNow:

—6

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

Private Sub Form_MouseUp (Button As Integer, _
        Shift As Integer, X As Single, Y As Single)
    DrawNow = False
End Sub
```

15

The MouseMove procedure draws a line only if DrawNow is True. Otherwise, it takes no action:

```
Private Sub Form_MouseMove (Button As Integer, _
        Shift As Integer, X As Single, Y As Single)
    If DrawNow Then Line -(X, Y)
End Sub
```

16

Each time the user presses a mouse button, the MouseDown event procedure is executed and turns drawing on. Then as the user holds the Mouse button down, the MouseMove event procedure is executed repeatedly as the pointer is dragged across the screen.

Note that the Line method omits the first endpoint, causing Visual Basic to start drawing at the mouse pointer's current coordinates. By default, the drawing coordinates correspond to the last point drawn; the form's CurrentX and CurrentY properties were reset in the Form_MouseDown procedure.

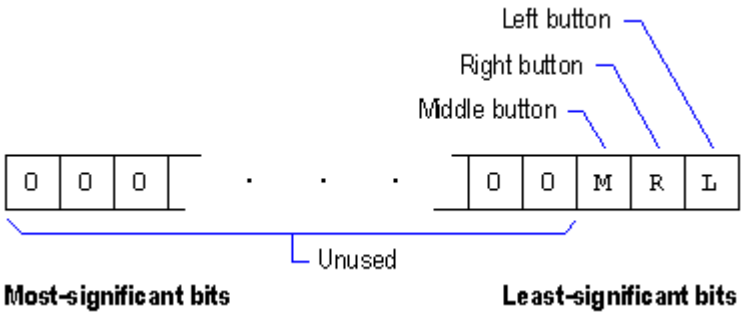
17

Detecting Mouse Buttons

You can make your applications more powerful by writing code that responds differently to mouse events, depending on which mouse button is used or whether the SHIFT, CTRL, or ALT key is pressed. To provide these options, you use the arguments *button* and *shift* with the MouseDown, MouseUp, and MouseMove event procedures. Techniques for using the *shift* argument are described in "Detecting SHIFT, CTRL , and ALT States" later in this chapter.

The MouseDown, MouseUp, and MouseMove events use the *button* argument to determine which mouse button or buttons are pressed. The *button* argument is a bit-field argument — a value in which each bit represents a state or condition. These values are expressed as integers. The three least-significant (lowest) bits represent the left, right, and middle mouse buttons, as shown in Figure 11.4.

Figure 11.4 How bits represent the state of the mouse



5

The default value of each bit is 0 (False). If no buttons are pressed, the binary value of the three bits is 000. If you press the left button, the binary value, or pattern, changes to 001. The left-button bit-value changes from 0 (False) to 1 (True).

The *button* argument uses either a decimal value or an constant to represent these binary patterns. The following table lists the binary value of the bits, the decimal equivalent, and the Visual Basic constant:

Binary Value	Decimal Value	Constant	Meaning
001	1	vbLeftButton	The left button is pressed.
010	2	vbRightButton	The right button is pressed.
100	4	vbMiddleButton	The middle button is pressed.

18

Note Visual Basic provides constants that represent the binary values of the *button* and *shift* arguments. These constants can be used interchangeably with their equivalent decimal values. Not all values have corresponding constants, however. The values for some button and/or shift combinations are derived by simply adding decimal values.

19

The middle button is assigned to decimal value 4. Pressing the left and right buttons simultaneously produces a single digit value of 3 (1+2). On a three-button mouse, pressing all three buttons simultaneously produces the decimal value of 7 (4+2+1). The following table lists the remaining button values derived from the possible button combinations:

Binary Value	Decimal Value	Constant	Meaning
000	0		No buttons are pressed.
011	3	vbLeftButton + vbRightButton	The left and right buttons are pressed.
101	5	vbLeftButton +	The left and middle

		vbMiddleButton	buttons are pressed.
110	6	vbRightButton + vbMiddleButton	The right and middle buttons are pressed.
111	7	vbRightButton + vbMiddleButton + vbLeftButton	All three buttons are pressed.

20

Using Button with MouseDown and MouseUp

You use the *button* argument with MouseDown to determine which button is being pressed and with MouseUp to determine which button has been released. Because only one bit is set for each event, you can't test for whether two or more buttons are being used at the same time. In other words, MouseDown and MouseUp only recognize one button press at a time.

Note In contrast, you can use the MouseMove event to test for whether two or more buttons are being pressed simultaneously. You can also use MouseMove to test for whether a particular button is being pressed, regardless of whether or not another button is being pressed at the same time. For more information, see "Using Button with MouseMove" later in this chapter.

21

You can specify which button causes a MouseDown or MouseUp event with simple code. The following procedure tests whether *button* equals 1, 2, or 4:

```
Private Sub Form_MouseDown (Button As Integer, _
        Shift As Integer, X As Single, Y As Single)
    If Button = 1 Then Print "You pressed _
        the left button."
    If Button = 2 Then Print "You pressed _
        the right button."
    If Button = 4 Then Print "You pressed _
        the middle button."
End Sub
```

22

If the user presses more than one button, Visual Basic interprets that action as two or more separate MouseDown events. It sets the bit for the first button pressed, prints the message for that button, and then does the same for the next button. Similarly, Visual Basic interprets the release of two or more buttons as separate MouseUp events.

The following procedure prints a message when a pressed button is released:

```
Private Sub Form_MouseUp(Button As Integer, _
        Shift As Integer, X As Single, Y As Single)
    If Button = 1 Then Print "You released _
        the left button."
    If Button = 2 Then Print "You released _
        the right button."
    If Button = 4 Then Print "You released _
        the middle button."
End Sub
```

23

Using Button with MouseMove

For the MouseMove event, *button* indicates the complete state of the mouse buttons — not just which button caused the event, as with MouseDown and MouseUp. This additional information is provided because all, some, or none of the bits might be set. This compares with just one bit per event in the MouseDown and MouseUp procedures.

Testing for a Single Button

If you test MouseMove for equality to 001 (decimal 1), you're testing to see if *only* the left mouse button is being held down while the mouse is moved. If another button is held down with the left button, the following code doesn't print anything:

```
Private Sub Form_MouseMove (Button As Integer, _
        Shift As Integer, X As Single, Y As Single)
    If Button = 1 Then Print "You're pressing _
        only the left button."
End Sub
```

24

To test for whether a particular button is down, use the And operator. The following code prints the message for each button pressed, regardless of whether another button is pressed:

```
Private Sub Form_MouseMove (Button As Integer, _
        Shift As Integer, X As Single, Y As Single)
    If Button And 1 Then Print "You're pressing _
        the left button."
    If Button And 2 Then Print "You're pressing _
        the right button."
End Sub
```

25

Pressing both buttons simultaneously prints both messages to the form. The MouseMove event recognizes multiple button states.

Testing for Multiple Buttons

In most cases, to isolate which button or buttons are being pressed, you use the MouseMove event.

Building on the previous examples, you can use the If…Then…Else statement to determine whether the left, right, or both buttons are being pressed. The following example tests for the three button states (left button pressed, right button pressed, and both buttons pressed) and prints the corresponding message.

Add the following code to the form's MouseMove event:

```
Private Sub Form_MouseMove(Button As Integer, _
        Shift As Integer, X As Single, Y As Single)
    If Button = 1 Then
        Print "You're pressing the left button."
    ElseIf Button = 2 Then
```

```
        Print "You're pressing the right button."
    ElseIf Button = 3 Then
        Print "You're pressing both buttons."
    End If
End Sub
```

26

You could also use the And operator with the Select Case statement to determine *button* and *shift* states. The And operator combined with the Select Case statement isolates the possible button states of a three-button mouse and then prints the corresponding message.

Create a variable called ButtonTest in the Declarations section of the form:

```
Dim ButtonTest as Integer
```

27

Add the following code to the form's MouseMove event:

```
Private Sub Form_MouseMove(Button As Integer, _
        Shift As Integer, X As Single, Y As Single)
ButtonTest = Button And 7
    Select Case ButtonTest
        Case 1 ' or vbLeftButton
            Print "You're pressing the left button."
        Case 2 ' or vbRightButton
            Print "You're pressing the right button."
        Case 4 ' or vbMiddleButton
            Print "You're pressing the middle button."
        Case 7
            Print "You're pressing all three buttons."
    End Select
End Sub
```

28

Using Button to Enhance Graphical Mouse Applications

You can use the *button* argument to enhance the Scribble application described in "The MouseMove Event" earlier in this chapter. In addition to drawing a continuous line when the left mouse button is pressed and stopping when the button is released, the application can draw a straight line from the last point drawn when the user presses the right button.

When writing code, it is often helpful to note each relevant event and the desired response. The three relevant events here are the mouse events:

- Form_MouseDown: This event takes a different action depending on the state of the mouse buttons: If the left button is down, set DrawNow to True and reset drawing coordinates; If the right button is down, draw a line.
- Form_MouseUp: If the left button is up, set DrawNow to False.
- Form_MouseMove: If DrawNow is True, draw a line.

6

The variable DrawNow is declared in the Declarations section of the form:

```
Dim DrawNow As Boolean
```

29

The MouseDown procedure has to take a different action, depending on whether the left or right mouse button caused the event:

```
Private Sub Form_MouseDown (Button As Integer, _
        Shift As Integer, X As Single, Y As Single)
    If Button = vbLeftButton Then
        DrawNow = True
        CurrentX = X
        CurrentY = Y
    ElseIf Button = vbRightButton Then
        Line -(X, Y)
    End If
End Sub
```

30

The following MouseUp procedure turns off drawing only when the left button is released:

```
Private Sub Form_MouseUp (Button As Integer, _
        Shift As Integer, X As Single, Y As Single)
    If Button = vbLeftButton Then DrawNow = False
End Sub
```

31

Note that within the MouseUp procedure, a bit set to 1 (vbLeftButton) indicates that the corresponding mouse button is released and drawing is turned off.

The following MouseMove procedure is identical to the one in the version of the Scribble application found in "The MouseMove Event" earlier in this chapter.

```
Private Sub Form_MouseMove (Button As Integer, _
        Shift As Integer, X As Single, Y As Single)
    If DrawNow Then Line -(X, Y)
End Sub
```

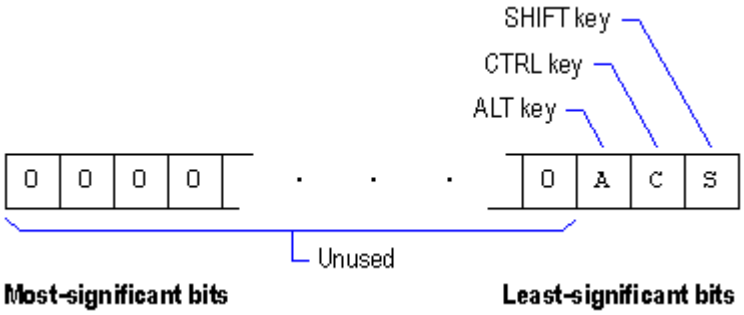
32

Detecting SHIFT, CTRL, and ALT States

The mouse and keyboard events use the *shift* argument to determine whether the SHIFT, CTRL, and ALT keys are pressed and in what, if any, combination. If the SHIFT key is pressed, *shift* is 1; if the CTRL key is pressed, *shift* is 2; and if the ALT key is pressed, *shift* is 4. To determine combinations of these keys, use the total of their values. For example, if SHIFT and ALT are pressed, *shift* equals 5 (1 + 4).

The three least-significant bits in *shift* correspond to the state of the SHIFT, CTRL, and ALT keys, as shown in Figure 11.5.

Figure 11.5 How bits represent the state of the SHIFT, CTRL, and ALT keys



7

Any or all of the bits in *shift* can be set, depending on the state of the SHIFT, CTRL, and ALT keys. These values and constants are listed in the following table:

Binary Value	Decimal Value	Constant	Meaning
001	1	vbShiftMask	The SHIFT key is pressed.
010	2	vbCtrlMask	The CTRL key is pressed.
100	4	vbAltMask	The ALT key is pressed.
011	3	vbShiftMask + vbCtrlMask	The SHIFT and CTRL keys are pressed.
101	5	vbShiftMask + vbAltMask	The SHIFT and ALT keys are pressed.
110	6	vbCtrlMask + vbAltMask	The CTRL and ALT keys are pressed.
111	7	vbCtrlMask + vbAltMask + vbShiftMask	The SHIFT, CTRL, and ALT keys are pressed.

33

As with the mouse events' *button* argument, you can use the If…Then…Else statement or the And operator combined with the Select Case statement to determine whether the SHIFT, CTRL, or ALT keys are being pressed and in what, if any, combination.

Open a new project and add the variable ShiftTest to the Declarations section of the form:

```
Dim ShiftTest as Integer
```

34

Add the following code to the form's MouseDown event:

```
Private Sub Form_MouseDown(Button As Integer, _
        Shift As Integer, X As Single, Y As Single)
    ShiftTest = Shift And 7
    Select Case ShiftTest
        Case 1 ' or vbShiftMask
```

```
        Print "You pressed the SHIFT key."
    Case 2 ' or vbCtrlMask
        Print "You pressed the CTRL key."
    Case 4 ' or vbAltMask
        Print "You pressed the ALT key."
    Case 3
        Print "You pressed both SHIFT and CTRL."
    Case 5
        Print "You pressed both SHIFT and ALT."
    Case 6
        Print "You pressed both CTRL and ALT."
    Case 7
        Print "You pressed SHIFT, CTRL, and ALT."
    End Select
End Sub
```

35

Dragging and Dropping

When you design Visual Basic applications, you often drag controls around on the form. The drag-and-drop features in Visual Basic allow you to extend this ability to the user at run time. The action of holding a mouse button down and moving a control is called *dragging,* and the action of releasing the button is called *dropping*.

Note Dragging a control at run time doesn't automatically change its location — you must program the relocation yourself, as described in "Changing the Position of a Control." Often, dragging is used only to indicate that some action should be performed; the control retains its original position after the user releases the mouse button.

36

Using the following drag-and-drop properties, events, and method, you can specify both the meaning of a drag operation and how dragging can be initiated (if at all) for a given control.

Category	Item	Description
Properties	DragMode	Enables automatic or manual dragging of a control.
	DragIcon	Specifies what icon is displayed when the control is dragged.
Events	DragDrop	Recognizes when a control is dropped onto the object.
	DragOver	Recognizes when a control is dragged over the object.
Methods	Drag	Starts or stops manual dragging.

37

All controls except menus, timers, lines, and shapes support the DragMode and DragIcon properties and the Drag method. Forms recognize the DragDrop and DragOver events, but they don't support the DragMode and DragIcon properties or the Drag method.

Note Controls can only be dragged when they do not have the focus. To prevent a control from getting the focus, set its TabStop property to False.

38

Enabling Automatic Drag Mode

To allow the user to drag a control, set its DragMode property to 1-Automatic.

When you set dragging to Automatic, dragging is always "on." For more control over dragging operations, use the 0-Manual setting described in "Controlling When Dragging Starts or Stops" later in this chapter.

Note While an automatic drag operation is taking place, the control being dragged doesn't recognize other mouse events.

39

Changing the Drag Icon

When dragging a control, Visual Basic uses a gray outline of the control as the default drag icon. You can substitute other images for the outline by setting the DragIcon property. This property contains a Picture object that corresponds to a graphic image.

The easiest way to set the DragIcon property is to use the Properties window. Select the DragIcon property, and then click the Properties button to select a file containing a graphic image from the Load Icon dialog box.

You can assign icons to the DragIcon property from the Icon Library included with Visual Basic. (The icons are located in the \Program files\Microsoft Visual Basic\Icons directory.) You can also create your own drag icons with a graphics program.

At run time, you can select a drag icon image by assigning the DragIcon property of one control to the same property of another:

```
Set Image1.DragIcon = Image2.DragIcon
```

40

You can also set the DragIcon property at run time by assigning the Picture property of one control to the DragIcon property of another:

```
Set Image1.DragIcon = Image3.Picture
```

41

Or, you can use the LoadPicture function:

```
Set Image1.DragIcon = LoadPicture("c:\Program _
    files\Microsoft Visual Basic\Icons\Disk04.ico")
```

42

For More Information For information on the Picture property and the LoadPicture function, see "Working with Text and Graphics."

43

Responding When the User Drops the Object

When the user releases the mouse button after dragging a control, Visual Basic generates a DragDrop event. You can respond to this event in many ways. Remember

that the control doesn't automatically move to the new location, but you can write code to relocate the control to the new location (indicated by the last position of the gray outline). See "Changing the Position of a Control" for more information.

Two terms are important when discussing drag-and-drop operations: *source* and *target*.

Term	Meaning
Source	The control being dragged. This control can be any object except a menu, timer, line, or shape.
Target	The object onto which the user drops the control. This object, which can be a form or control, recognizes the DragDrop event.

44

A control becomes the target if the mouse position is within its borders when the button is released. A form is the target if the pointer is in a blank portion of the form.

The DragDrop event provides three arguments: *source*, *x*, and *y*. The *source* argument is a reference to the control that was dropped onto the target.

Because *source* is declared As Control, you use it just as you would a control — you can refer to its properties or call one of its methods.

The following example illustrates how the source and target interact. The source is an Image control with its Picture property set to load a sample icon file representing a few file folders. Its DragMode property has been set to 1-Automatic and its DragIcon property to a sample drag-and-drop icon file. The target, also an image control, contains a picture of an open file cabinet.

Add the following procedure to the second image control's DragDrop event:

```
Private Sub Image2_DragDrop(Source As Control, _
        X As Single, Y As Single)
    Source.Visible = False
    Image2.Picture = LoadPicture("c:\Program _
        Files\Microsoft Visual _
        Basic\Icons\Office\Files03a.ico")
End Sub
```

45

Dragging and dropping Image1 onto Image2 causes Image1 to vanish and Image2 to change its picture to that of a closed file cabinet. Using the *source* argument, the Visible property of Image1 was changed to False.

Note You should use the *source* argument carefully. Although you know that it always refers to a control, you don't necessarily know which type of control. For example, if the control is a text box and you attempt to refer to Source.Value, the result is a run-time error because text boxes have no Value property.

46

You can use the If...Then...Else statement with the TypeOf keyword to determine what kind of control was dropped.

For More Information See "If…Then…Else" in "Programming with Objects."

47

Controlling When Dragging Starts or Stops

Visual Basic has a Manual setting for the DragMode property that gives you more control than the Automatic setting. The Manual setting allows you to specify when a control can and cannot be dragged. (When DragMode is set to Automatic, you can always drag the control as long as the setting isn't changed.)

For instance, you may want to enable dragging in response to MouseDown and MouseUp events, or in response to a keyboard or menu command. The Manual setting also allows you to recognize a MouseDown event before dragging starts, so that you can record the mouse position.

To enable dragging from code, leave DragMode in its default setting (0-Manual). Then use the Drag method whenever you want to begin or stop dragging an object. Use the following Visual Basic constants to specify the *action* of the Drag argument.

Constant	Value	Meaning
vbCancel	0	Cancel drag operation
vbBeginDrag	1	Begin drag operation
vbEndDrag	2	End drag operation

48

The syntax for the Drag method is as follows:

[*object*.]**Drag** *action*

49

If *action* is set to vbBeginDrag, the Drag method initiates dragging of the control. If *action* is set to vbEndDrag, the control is dropped, causing a DragDrop event. If *action* is set to vbCancel, the drag is canceled. The effect is similar to giving the value vbEndDrag, except that no DragDrop event occurs.

Building on the example given in "Responding When the User Drops the Object" earlier in this chapter, you can add a MouseDown event for Image1 that illustrates the Drag method. Set the Image1 DragMode property to 0-Manual, then add the following procedure:

```
Private Sub Image1_MouseDown(Button As Integer, _
        Shift As Integer, X As Single, Y As Single)
    Image1.Drag vbBeginDrag
    Set Image1.DragIcon = LoadPicture("c:\Program _
        files\ Microsoft Visual _
        Basic\Icons\Dragdrop\Dragfldr.ico")
End Sub
```

50

Adding a DragOver event procedure to Image2 allows you to terminate dragging when the source enters the target. This example closes the file cabinet when Image1 is passed over Image2.

```
Private Sub Image2_DragOver(Source As Control, _
```

```
        X As Single, Y As Single, State As Integer)
    Source.Drag vbEndDrag
    Source.Visible = False
    Image2.Picture = LoadPicture("c:\Program _
        files\Microsoft Visual _
        Basic\Icons\Office\Files03a.ico")
End Sub
```

51

Adding a third Image control to the form demonstrates canceling a drag operation. In this example the Image3 Picture property contains an icon of a trash can. Using the DragOver event and the *source* argument, dragging the files over Image3 cancels the drag operation.

```
Private Sub Image3_DragOver(Source As Control, _
        X As Single, Y As Single, State As Integer)
    Source.Drag vbCancel
End Sub
```

52

Changing the Position of a Control

You may want the source control to change position after the user releases the mouse button. To move a control to the new mouse location, use the Move method with any control that has been drag-enabled.

You can reposition a control when it is dragged and dropped to any location on the form not occupied by another control. To illustrate this, start a new Visual Basic project, add an Image control to the form and assign it any icon or bitmap by setting the Picture property, and then change the Image control's DragMode property to 1-Automatic.

Add the following procedure to the form's DragDrop event:

```
Private Sub Form_DragDrop (Source As Control, _
        X As Single, Y As Single)
    Source.Move X, Y
End Sub
```

53

This code may not produce precisely the effects you want, because the upper-left corner of the control is positioned at the mouse location. This code positions the center of the control at the mouse location:

```
Private Sub Form_DragDrop (Source As Control, _
        X As Single, Y As Single)
    Source.Move (X - Source.Width / 2), _
        (Y - Source.Height / 2)
End Sub
```

54

The code works best when the DragIcon property is set to a value other than the default (the gray rectangle). When the gray rectangle is being used, the user usually wants the control to move precisely into the final position of the gray rectangle. To do this, record the initial mouse position within the source control. Then use this position as an offset when the control is moved.

To record the initial mouse position

1 Specify manual dragging of the control.

2 Declare two form-level variables, DragX and DragY.

3 Turn on dragging when a MouseDown event occurs.

4 Store the value of *x* and *y* in the form-level variables in this event.

8

The following example illustrates how to cause drag movement for an image control named Image1. The control's DragMode property should be set to 0-Manual at design time. The Declarations section contains the form-level variables DragX and DragY, which record the initial mouse position within the Image control:

```
Dim DragX As Single, DragY As Single
```

55

The MouseDown and MouseUp procedures for the control turn dragging on and drop the control, respectively. In addition, the MouseDown procedure records the mouse position inside the control at the time dragging begins:

```
Private Sub Image1_MouseDown (Button As Integer, _
        Shift As Integer, X As Single, Y As Single)
    Image1.Drag 1
    DragX = X
    DragY = Y
End Sub
```

56

The Form_DragDrop procedure actually moves the control. To simplify this example, assume that Image1 is the only control on the form. The target can therefore only be the form itself. The Form_DragDrop procedure repositions the control, using DragX and DragY as offsets:

```
Private Sub Form_DragDrop (Source As Control, _
        X As Single, Y As Single)
    Source.Move (X - DragX), (Y - DragY)
End Sub
```

57

Note that this example assumes that Image1 and the form use the same units in their respective coordinate systems. If they don't, then you'll have to convert between units.

For More Information For information on coordinate systems, see "Working with Text and Graphics."

58

OLE Drag and Drop

One of the most powerful and useful features you can add to your Visual Basic applications is the ability to drag text or graphics from one control to another, or from a control to another Windows application, and vice versa. OLE drag-and-drop allows you to add this functionality to your applications.

With OLE drag and drop, you're not dragging one control to another control to invoke some code (as with the drag and drop discussed earlier in this chapter); you're

moving *data* from one control or application to another control or application. For example, the user selects and drags a range of cells in Excel, then drops the range of cells into the Data-Bound Grid control in your application.

Almost all Visual Basic controls support OLE drag-and-drop to some degree. The following standard and ActiveX controls (those provided in the Professional and Enterprise editions of Visual Basic) provide automatic support for OLE drag-and-drop, which means that no code needs to be written to either drag from or drop to the control:

Apex Data-Bound Grid	Picture box	Rich text box
Image	Text box	Masked edit box

59

To enable automatic OLE dragging and dropping for these controls, you set the OLEDragMode and OLEDropMode properties to Automatic.

Some controls only provide automatic support for the OLE drag operation. To enable automatic dragging from these controls, set the OLEDragMode property to Automatic.

Combo box	Data-Bound list box	File list box
Data-Bound Combo box	Directory list box	List box
Tree View	List View	

60

The following controls only support the OLE drag-and-drop events, meaning that you can program them with code to act either as the source or target of the OLE drag-and-drop operations.

Check box	Frame	Option button
Command button	Label	Drive list box
Data		

61

Note To determine if other ActiveX controls support OLE drag and drop, load the control into Visual Basic and check for the existence of the OLEDragMode and OLEDropMode properties, or for the OLEDrag method. (A control that does not have automatic support for OLE drag will not have the OLEDragMode property, but it will have an OLEDrag method if it supports OLE drag through code.)

62

Note Forms, MDI forms, Document Objects, User Controls, and Property Pages contain the OLEDropMode property and provide support for manual dragging and dropping only.

63

Using the following OLE drag-and-drop properties, events, and method, you can specify how a given control responds to dragging and dropping.

Category	Item	Description
Properties	OLEDragMode	Enables automatic or manual dragging of a

		control (if the control supports manual but not automatic OLE drag, it will not have this property but it will support the OLEDrag method and the OLE drag-and-drop events).
	OLEDropMode	Specifies how the control will respond to a drop.
Events	OLEDragDrop	Recognizes when a source object is dropped onto a control.
	OLEDragOver	Recognizes when a source object is dragged over a control.
	OLEGiveFeedback	Provides customized drag icon feedback to the user, based on the source object.
	OLEStartDrag	Specifies which data formats and drop effects (copy, move, or refuse data) the source supports when dragging is initiated.
	OLESetData	Provides data when the source object is dropped.
	OLECompleteDrag	Informs the source of the action that was performed when the object was dropped into the target.
Method	OLEDrag	Starts manual dragging.

64

Automatic vs. Manual Dragging and Dropping

It is helpful to think of OLE drag-and-drop implementation as either *automatic* or *manual*.

Automatic dragging and dropping means that, for example, you can drag text from one text box control to another by simply setting the OLEDragMode and OLEDropMode properties of these controls to Automatic: You don't need to write any code to respond to any of the OLE drag-and-drop events. When you drag a range of cells from Excel into a Word document, you've performed an automatic drag-and-drop operation. Depending upon how a given control or application supports OLE drag and drop and what type of data is being dragged, automatically dragging and dropping data may be the best and simplest method.

Manual dragging and dropping means that you have chosen (or have been forced to) manually handle one or more of the OLE drag-and-drop events. Manual implementation of OLE drag and drop may be the better method when you want to gain greater control over each step in the process, to provide the user with customized visual feedback, to create your own data format. Manual implementation is the only option when a control does not support automatic dragging and dropping.

It is also helpful to define the overall model of the OLE drag-and-drop operation. In a drag and drop operation, the object from which data is dragged is referred to as the *source.* The object into which the data is dropped is referred to as the *target*. Visual Basic provides the properties, events, and method to control and respond to actions affecting both the source and the target. It is also helpful to recognize that the source and the target may be in different applications, in the same application, or even in the

same control. Depending upon the scenario, you may need to write code for either the source or target, or both.

Enabling Automatic OLE Drag and Drop

If your controls support automatic dragging and dropping, you can drag data from and/or drop data into a Visual Basic control by setting the control's OLEDragMode and/or OLEDropMode properties to Automatic. For instance, you may want to drag text from a text box control into a Word for Windows document, or allow the text box control to accept data dragged from the Word for Windows document.

To allow dragging from the text box control, set the OLEDragMode property to Automatic. At run time, you can select text typed into the text box control and drag it into the open Word for Windows document.

When you drag text from the text box control into a Word for Windows document, it is, by default, moved rather than copied into the document. If you hold the CTRL key down while dropping text, it will be copied rather than moved. This is the default behavior for all objects or applications that support OLE drag-and-drop. To restrict this operation by allowing data to only be moved or only be copied, you need to modify the automatic behavior by using the manual dragging and dropping techniques. For more information, see "Using the Mouse and Keyboard to Modify Drop Effects and User Feedback."

To allow the text box control to automatically retrieve data in a OLE drag-and-drop operation, set its OLEDropMode property to Automatic. At run time, data dragged from an OLE-enabled application into the text box control will be moved rather than copied unless you hold down the CTRL key during the drop, or alter the default behavior through code.

Automatic support for dragging and dropping data has its limitations; some of these limitations are derived from the functionality of the controls themselves. For instance, if you move text from a Word for Windows document into a text box control, all the rich text formatting in the Word document will be stripped out because the text box control doesn't support this formatting. Similar limitations exist for most controls. Another limitation of automatic operations is that you don't have complete control over what kind of data is dragged and/or dropped.

Note When dragging data, you may notice that the mouse pointer indicates if the object that it is passing over supports OLE drag and drop for the type of data that you are dragging. If the object supports OLE drag and drop for the type of data, the "drop" pointer is displayed. If the object does not, a "no drop" pointer is displayed.

65

The OLE Drag and Drop DataObject Object

OLE drag-and-drop uses the same *source* and *target* model as the simple event-driven drag-and-drop techniques discussed in "Dragging and Dropping." In this case,

however, you're not dragging one control to another control to invoke some code; you're moving *data* from one control or application to another control or application. For example, the user selects and drags a range of cells in Excel (*source*) then drops the range of cells into the DBGrid control (*target*) in your application.

In Visual Basic, the vehicle, or repository, of this data is the DataObject object — it is the means by which data is moved from the source to the target. It does this by providing the methods needed to store, retrieve, and analyze the data. The following table lists the property and methods used by the DataObject object:

Category	Item	Description
Property	Files	Holds the names of files dragged to or from the Windows Explorer.
Methods	Clear	Clears the content of the DataObject object.
	GetData	Retrieves data from the DataObject object.
	GetFormat	Determines if a specified data format is available in the DataObject object.
	SetData	Places data into the DataObject object, or indicates that a specified format is available upon request.

66

Used with the OLE drag-and-drop events, these methods allow you to manage data in the DataObject object on both the source and target sides (if both are within your Visual Basic application). For instance, you can place data into the DataObject object on the source side using the SetData method, and then use the GetData method to accept the data on the target side.

The Clear method is used to clear the content of the DataObject object on the source side when the OLEStartDrag event is triggered. When data from a control is dragged in an automatic drag operation, its data formats are placed into the DataObject object before the OLEStartDrag event is triggered. If you don't want to use the default formats, you use the Clear method. If you want to add to the default data formats, you do not use the Clear method.

The Files property allows you to store the names of a range of files that can be then dragged into a drop target. See "Dragging Files from the Windows Explorer" for more information on this property.

You can also specify the format of the data being transferred. The SetData and GetData methods use the following arguments to place or retrieve data in the DataObject object:

Argument	Description
data	Allows you to specify the type of data that is placed into the DataObject object (optional argument if the *format* argument has been set; otherwise, it's required).
format	Allows you to set several different formats that the source can support,

without having to load the data for each (optional argument if the *data* argument has been set or if Visual Basic understands the format; otherwise, it's required).

67

Note When data is dropped onto the target and no format has been specified, Visual Basic is able to detect if it is a bitmap, metafile, enhanced metafile, or text. All other formats must be specified explicitly or an error will be generated.

68

The *format* argument uses the following constants or values to specify the format of the data:

Constant	Value	Meaning
vbCFText	1	Text
vbCFBitmap	2	Bitmap (.bmp)
vbCFMetafile	3	Metafile (.wmf)
vbCFEMetafile	14	Enhanced metafile (.emf)
vbCFDIB	8	Device-independent bitmap (.dib or .bmp)
vbCFPalette	9	Color palette
vbCFFiles	15	List of files
vbCFRTF	-16639	Rich text format (.rtf)

69

The SetData, GetData, and GetFormat methods use the *data* and *format* arguments to return either the type of data in the DataObject object or to retrieve the data itself if the format is compatible with the *target*. For example:

```
Private Sub txtSource_OLEStartDrag(Data As _
        VB.DataObject, AllowedEffects As Long)
    Data.SetData txtSource.SelText, vbCFText
End Sub
```

70

In this example, *data* is the text selected in a textbox and *format* has been specified as text (vbCFText).

Note You should use the vbCFDIB data format instead of vbCFBitmap and vbCFPalette, in most cases. The vbCFDIB format contains both the bitmap and palette and is therefore the preferred method of transferring a bitmap image. You can, however, also specify the vbCFBitmap and vbCFPalette for completeness. If you chose not to use the vbCFDIB format, you must specify both the vbCFBitmap and vbCFPalette formats so that the bitmap and the palette are correctly placed into the DataObject object.

71

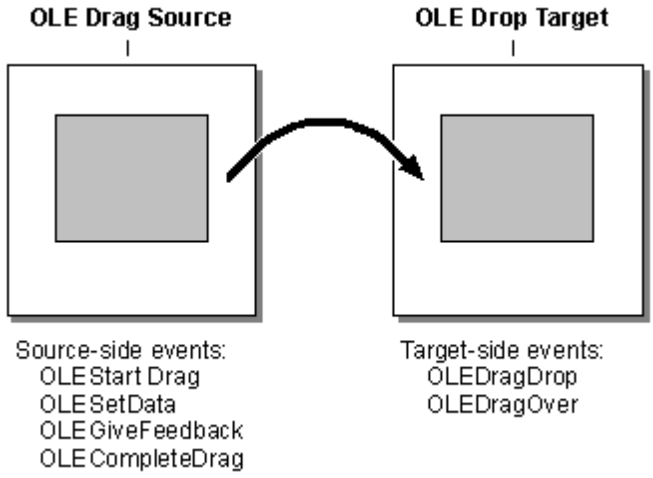
For More Information See “Creating a Custom Data Format” for information on defining your own data format.

72

How OLE Drag and Drop Works

When an OLE drag-and-drop operation is performed, certain events are generated on the source and target sides. The events associated with the source object are always generated, whether the drag-and-drop operation is automatic or manual. The target-side events, however, are only generated in a manual drop operation. The following illustration shows which events occur and can be responded to on the drag source, and which occur and can be responded to on the drop target.

Figure 11.6 Source-side and target-side events



9

Which events you'll need to respond to depends upon how you've chosen to implement the drag-and-drop functionality. For example, you may have created an application with a text box that you want to allow to automatically accept dragged data from another application. In this case, you simply set the control's OLEDropMode property to Automatic. If you want to allow data to be automatically dragged from the text box control as well, you set its OLEDragMode property to Automatic.

If, however, you want to change the default mouse cursors or enhance the functionality for button states and shift keys, you need to manually respond to the source- and target-side events. Likewise, if you want to analyze the data before it is dropped into a control (to verify that the data is compatible, for instance), or delay when the data is loaded into the DataObject object (so that multiple formats don't need to be loaded at the beginning), you'll need to use manual OLE drag-and-drop operations.

Because you can drag and drop data into numerous Visual Basic controls and Windows applications — with varying limitations and requirements — implementing OLE drag and drop can range from straightforward to fairly complex. The simplest implementation, of course, would be dragging and dropping between two automatic objects, whether the object is a Word document, an Excel spreadsheet, or a control in

your application that has been set to Automatic. Specifying multiple data formats that would be acceptable to your drop target would be more complicated.

Starting the Drag

What happens in a basic manual OLE drag-and-drop operation within your Visual Basic application? When the user drags data from an OLE drag source (a text box control, for example) by selecting and then holding down the left mouse button, the OLEStartDrag event is triggered and you can then either store the data or simply specify the formats that the source supports. You also need to specify whether copying or moving the data, or both, is allowed by the source.

For More Information See "Starting the OLE Drag Operation" for more information on the OLEDrag method, the OLEstartDrag event, using the SetData method to specify the supported data formats, and placing data into the DataObject.

73

Dragging Over the Target

As the user drags over the target, the target's OLEDragOver event is triggered, indicating that the source is within its boundaries. You then specify what the target would do if the data were dropped there — either copy, move, or refuse the data. By convention, the default is usually move, but it may be copy.

When the target specifies which drop effect will be performed if the source is dropped there, the OLEGiveFeedback event is triggered. The OLEGiveFeedback event is used to provide visual feedback to the user on what action will be taken when the selection is dropped — i.e., the mouse pointer will be changed to indicate a copy, move, or "no drop" action.

As the source is moved around within the boundaries of the target — or if the user presses the SHIFT, CTRL, or ALT keys while holding down the mouse button — the drop effect may be changed. For example, instead of allowing a copy or a move, the data may be refused.

If the user passes beyond the target or presses the ESC key, for example, then the drag operation may be canceled or modified (the mouse pointer may be changed to indicate that the object it is currently passing over will not accept the data).

For More Information See "Providing the User with Visual Feedback" and "Dragging the OLE Drag Source over the OLE Drop Target" for more information on the OLEDragOver and OLEGiveFeedback events.

74

Completing the Drag

When the user drops the source onto the target, the target's OLEDragDrop event is triggered. The target queries the source for the format of the data it contains (or supports, if the data wasn't placed into the source when the drag was started) and then either retrieves or rejects the data.

If the data was stored when the drag started, the target retrieves the data by using the GetData method. If the data wasn't stored when the drag started, the data is retrieved by triggering the source's OLESetData event and then using the SetData method.

When the data is accepted or rejected, the OLECompleteDrag event is triggered and the source can then take the appropriate action: if the data is accepted and a move is specified, the source deletes the data, for example.

For More Information See "Dropping the OLE Drag Source onto the OLE Drop Target" for more information on the OLEDragDrop event, the OLECompleteDrag event, and using the GetFormat and GetData methods to retrieve data from the DataObject object.

75

Starting the OLE Drag Operation

If you want to be able to specify which data formats or drop effects (copy, move, or no drop) are supported, or if the control you want to drag from doesn't support automatic dragging, you need to make your OLE drag operation manual.

The first phase of a manual drag-and-drop operation is calling the OLEDrag method, setting the allowed drop effects, specifying the supported data formats, and, optionally, placing data into the DataObject object.

You use the OLEDrag method to manually start the drag operation and the OLEStartDrag event to specify the allowed drop-action effects and the supported data formats.

The OLEDrag Method

Generally, the OLEDrag method is called from an object's MouseMove event when data has been selected, the left mouse button is pressed and held, and the mouse is moved.

The OLEDrag method does not provide any arguments. Its primary purpose is to initiate a manual drag and then allow the OLEStartDrag event to set the conditions of the drag operation (for example, specifying what will happen when the data is dragged into another control).

If the source control supports the OLEDragMode property, to have manual control over the drag operation you must set the property to Manual and then use the OLEDrag method on the control. If the control supports manual but not automatic OLE drag, it will not have the OLEDragMode property, but it will support the OLEDrag method and the OLE drag-and-drop events.

Note The OLEDrag method will also work if the source control's OLEDragMode property is set to Automatic.

76

Specifying Drop Effects and Data Formats

In a manual OLE drag operation, when the user begins dragging the source and the OLEDrag method is called, the control's OLEStartDrag event fires. Use this event to specify what drop effects and data formats the source supports.

The OLEStartDrag event uses two arguments to specify supported data formats and whether the data can be copied or moved when the data is dropped (drop effects).

Note If no drop effects or data formats are specified in the OLEStartDrag event, the manual drag will not be started.

77

The AllowedEffects Argument

The *allowedeffects* argument specifies which drop effects the drag source supports. For example:

```
Private Sub txtSource_OLEStartDrag(Data As _
        VB.DataObject, AllowedEffects As Long)
    AllowedEffects = vbDropEffectMove Or _
        vbDropEffectCopy
End Sub
```

78

The target can then query the drag source for this information and respond accordingly.

The *allowedeffects* argument uses the following values to specify drop effects:

Constant	Value	Description
vbDropEffectNone	0	Drop target cannot accept the data.
vbDropEffectCopy	1	Drop results in a copy. The original data is untouched by the drag source.
VbDropEffectMove	2	Drag source removes the data.

79

The Format Argument

You specify which data formats the object supports by setting the *format* argument of the OLEStartDrag event. To do this, you use the SetData method. For example, in a scenario using a rich text box control as a source and a text box control as a target, you might specify the following supported formats:

```
Private Sub rtbSource_OLEStartDrag(Data As _
        VB.DataObject, AllowedEffects As Long)
    AllowedEffects = vbDropEffectMove Or _
        vbDropEffectCopy

    Data.SetData , vbCFText
    Data.SetData , vbCFRTF
End Sub
```

80

The target can query the source to determine which data formats are supported and then respond accordingly — e.g., if the format of the dropped data is not supported by the target, reject the dropped data. In this case, the only data formats that are supported by the source are the text and rich-text formats.

For More Information See “The OLE Drag and Drop Data Object" for more information on format values for the SetData method.

81

Placing Data into the DataObject object

In many cases, especially if the source supports more than one format, or if it is time-consuming to create the data, you may want to place data into the DataObject object only when it is requested by the target. You can, however, place the data into the DataObject object when you begin a drag operation by using the SetData method in the OLEStartDrag event. For example:

```
Private Sub txtSource_OLEStartDrag(Data As _
        VB.DataObject, AllowedEffects As Long)
    Data.Clear
    Data.SetData txtSource.SelText, vbCFText
End Sub
```

82

This example clears the default data formats from the DataObject object using the Clear method, specifies the data format (text) of the selected data, and then places the data into the DataObject object with the SetData method.

Dragging the OLE Drag Source over the OLE Drop Target

With a manual target, you can determine and respond to the position of the source data within the target and respond to the state of the mouse buttons and the SHIFT, CTRL, and ALT keys. Where both the source and the target are manual, you can modify the default visual behavior of the mouse.

To . . .	Use the . . .
Determine and respond to the position of the source object	*state* argument of the OLEDragOver event
Respond to the state of the mouse buttons	*button* argument of the OLEDragDrop and OLEDragOver events
Respond to the state of the SHIFT, CTRL, and ALT keys	*shift* arguments of the OLEDragDrop and OLEDragOver events
Modify the default visual behavior of the mouse	*effect* argument of the OLEDragOver event and the *effect* argument of the OLEGiveFeedback

83

For More Information For more information about changing the mouse cursor, see "Providing the User with Customized Visual Feedback." For more information about using the *button* and *shift* arguments, see "Using the Mouse and Keyboard to Modify Drop Effects and User Feedback."

84

The OLEDragOver Event State Argument

Depending upon its position, the *effect* argument may be changed to indicate the currently acceptable drop effect.

The *state* argument of the OLEDragOver event allows you to respond to the source data entering, passing over, and leaving the target control. For example, when the source data enters the target control, the *state* argument is set to vbEnter.

When the drag source is moved around within the boundaries of the drop target, the *state* argument is set to vbOver. Depending upon the position (the *x* and *y* arguments) of the mouse pointer, you may want to change the drag effect. Notice that the OLEDragOver event is generated several times a second, even when the mouse is stationary.

The *state* argument of the OLEDragOver event specifies when the data enters, passes over, and leaves the target control by using the following constants:

Constant	Value	Meaning
vbEnter	0	Data has been dragged within the range of a target.
vbLeave	1	Data has been dragged out of the range of a target.
vbOver	2	Data is still within the range of a target, and either the mouse has moved, a mouse or keyboard button has changed, or a certain system-determined amount of time has elapsed.

Providing the User with Customized Visual Feedback

If you want to modify the default visual behavior of the mouse in an OLE drag-and-drop operation, you can manipulate the OLEDragOver event on the target side and the OLEGiveFeedback event on the source side.

OLE drag and drop provides automatic visual feedback during a drag-and-drop operation. For example, when you start a drag, the mouse pointer is changed to indicate that a drag has been initiated. When you pass over objects that do not support OLE drop, the mouse pointer is changed to the "no drop" cursor.

Modifying the mouse pointer to indicate how a control will respond if the data is dropped onto it involves two steps: determining what type of data is in the DataObject object using the GetFormat method, and then setting the *effect* argument of the OLEDragOver event to inform the source what drop effects are allowed for this control.

The OLEDragOver Event

When a target control's OLEDropMode property is set to Manual, the OLEDragOver event is triggered whenever dragged data passes over the control.

The *effect* argument of the OLEDragOver event is used to specify what action would be taken if the object were dropped. When this value is set, the source's

OLEGiveFeedback event is triggered. The OLEGiveFeedback event contains its own *effect* argument, which is used to provide visual feedback to the user on what action will be taken when the selection is dragged — i.e., the mouse pointer is changed to indicate a copy, move, or "no drop" action.

The *effect* argument of the OLEDragOver event uses the following constants to indicate the drop action:

Constant	Value	Description
vbDropEffectNone	0	Drop target cannot accept the data.
vbDropEffectCopy	1	Drop results in a copy. The original data is untouched by the drag source.
VbDropEffectMove	2	Drag source removes the data.

85

Note The *effect* argument of the OLEDragOver and OLEGiveFeedback events express the same drop effects (copy, move, no drop) as the *allowedeffects* argument of the OLEStartDrag event. They differ only in that the OLEStartDrag event specifies which effects are allowed, and the OLEDragOver and OLEGiveFeedback use the *effect* argument to indicate to the source which of these actions will be taken.

86

The following code example queries the DataObject object for a compatible data format for the target control. If the data is compatible, the *effect* argument informs the source that a move will be performed if the data is dropped. If the data is not compatible, the source will be informed and a "'no drop" mouse pointer will be displayed.

```
Private Sub txtTarget_OLEDragOver(Data As _
        VB.DataObject, Effect As Long, Button As _
        Integer, Shift As Integer, X As Single, _
        Y As Single, State As Integer)
    If Data.GetFormat(vbCFText) Then
        Effect = vbDropEffectMove And Effect
    Else
        Effect = vbDropEffectNone
    End If
End Sub
```

87

When the source data is dragged over the target, and the OLEDragOver event is triggered, the source tells the target which effects it allows (move, copy, no drop). You must then chose which single effect will occur if the data is dropped. The *effect* argument of the OLEDragOver event informs the source which drop action it supports, and the source then informs the user by using the OLEGiveFeedback event to modify the mouse pointer.

The OLEGiveFeedback Event

To change the default behavior of the mouse pointer based on the *effect* argument of the OLEDragOver event, you need to manually specify new mouse pointer values

using the OLEGiveFeedback event. The source's OLEGiveFeedback event is triggered automatically when the *effect* argument of the OLEDragOver event is set.

The OLEGiveFeedback event contains two arguments (*effect* and *defaultcursors*) that allow you to modify the default mouse pointers in an OLE drag-and-drop operation.

The *effect* argument, like the other OLE drag-and-drop events, specifies whether data is to be copied, moved, or rejected. The purpose of this argument in the OLEGiveFeedback event, however, is to allow you to provide customized visual feedback to the user by changing the mouse pointer to indicate these actions.

Constant	Value	Description
vbDropEffectNone	0	Drop target cannot accept the data.
vbDropEffectCopy	1	Drop results in a copy. The original data is untouched by the drag source.
vbDropEffectMove	2	Drag source removes the data.
VbDropEffectScroll	&H80000000&	Scrolling is about to start or is currently occuring in the target. The value is used in addition to the other values.

88

Note The vbDropEffectScroll value can be used by some applications or controls to indicate that the user is causing scrolling by moving the mouse pointer near the edge of an application's window. Scrolling is automatically supported by some but not all of the Visual Basic standard controls. You may need to program for the scroll effect if you drag data into a program that contains scroll bars — Word for Windows, for example.

89

The *defaultcursors* argument specifies whether the default OLE cursor set is used. Setting this argument to False allows you to specify your own cursors using the Screen.MousePointer property of the Screen object.

In most cases, specifying custom mouse pointers is unnecessary because the default behavior of the mouse is handled by OLE. If you decide to specify custom mouse pointers using the OLEGiveFeedback event, you need to account for every possible effect, including scrolling. It is also a good idea to program for effects that may be added later by creating an option that gives the control of the mouse pointer back to OLE if an unknown effect is encountered.

The following code example sets the *effect* and *defaultcursors* arguments and specifies custom cursors (.ico or .cur files) for the copy, move, and scroll effects by setting the MousePointer and MouseIcon properties of the Screen object. It also returns control of the mouse pointer back to OLE if an unknown effect is encountered.

```
Private Sub TxtSource_OLEGiveFeedback(Effect As Long, _
        DefaultCursors As Boolean)
    DefaultCursors = False
    If Effect = vbDropEffectNone Then
```

```
        Screen.MousePointer = vbNoDrop
    ElseIf Effect = vbDropEffectCopy Then
        Screen.MousePointer = vbCustom
        Screen.MouseIcon = LoadPicture("c:\copy.ico")
    ElseIf Effect = (vbDropEffectCopy Or _
        vbDropEffectScroll) Then
            Screen.MousePointer = vbCustom
            Screen.MouseIcon = _
                LoadPicture("c:\copyscrl.ico")
    ElseIf Effect = vbDropEffectMove Then
        Screen.MousePointer = vbCustom
        Screen.MouseIcon = LoadPicture("c:\move.ico")
        ElseIf Effect = (vbDropEffectMove Or _
            vbDropEffectScroll) Then
            Screen.MousePointer = vbCustom
            Screen.MouseIcon = _
LoadPicture("c:\movescrl.ico")
        Else
            ' If some new format is added that we do not
            '     understand, allow OLE to handle it with
            '     correct defaults.
            DefaultCursors = True
    End If
End Sub
```

90

Note You should always reset the mouse pointer in the OLECompleteDrag event if you specify a custom mouse pointer in the OLEGiveFeedback event. See “Informing the Source When Data is Dropped” for more information.

91

For More Information See “Customizing the Mouse Pointer” for information on setting the MousePointer and MouseIcon properties.

92

Dropping the OLE Drag Source onto the OLE Drop Target

If your target supports manual OLE drag-and-drop operations, you can control what happens when the cursor is moved within the target and can specify what kind of data the target will accept. When the user drops the source object onto the target control, the OLEDragDrop event is used to query the DataObject object for a compatible data format, and then retrieve the data.

The OLEDragDrop event also informs the source of the drop action, allowing it to delete the original data if a move has been specified, for example.

Retrieving the Data

The OLEDragDrop event is triggered when the user drops the source onto the target. If data was placed into the DataObject object when the drag operation was initiated, it can be retrieved when the OLEDragDrop event is triggered, by using the GetData method. If, however, only the supported source formats were declared when the drag

operation was initiated, then the GetData method will automatically trigger the OLESetData event on the source to place the data into, and then retrieve the data from, the DataObject object.

The following example retrieves data that was placed into the DataObject object when the drag operation was initiated. The drag operation may have been initiated manually (using the OLEDrag method on the source) or automatically (by setting the OLEDragMode property of the source to Automatic). The dragged data is retrieved using the DataObject object's GetData method. The GetData method provides you with constants that represent the data types that the DataObject object supports. In this case, we are retrieving the data as text.

```
Private Sub txtTarget_OLEDragDrop(Data As _
        VB.DataObject, Effect As Long, Button As _
        Integer, Shift As Integer, X As Single, _
        Y As Single)
    txtTarget.Text = Data.GetData(vbCFText)
End Sub
```

93

For More Information For a complete list of GetData format constants, see "The OLE Drag and Drop DataObject Object" earlier in this chapter.

94

Querying the DataObject Object

You may need to query the DataObject object for the types of data that are being dropped onto the target. You use the GetFormat method in an If…Then statement to specify which types of data the target control can accept. If the data within the DataObject object is compatible, the drop action will be completed.

```
Private Sub txtTarget_OLEDragDrop(Data As _
        VB.DataObject, Effect As Long, Button As _
        Integer, Shift As Integer, X As Single, _
        Y As Single)
    If Data.GetFormat(vbCFText) Then
        txtTarget.Text = Data.GetData(vbCFText)
    End If
End Sub
```

Placing Data into the DataObject Object

When the target uses the GetData method to retrieve data from the source, the OLESetData event is only triggered if the data was not placed into the source when the drag operation was initiated.

In many cases, especially if the source supports more than one format, or if it is time-consuming to create the data, you may want to place data into the DataObject object only when it is requested by the target. The OLESetData event allows the source to respond to only one request for a given format of data.

For example, if the supported data formats were specified using the OLEStartDrag event when the drag operation was initiated, but data was not placed into the

DataObject object, the OLESetData event is used to place a specific format of data into the DataObject object.

```
Private Sub txtSource_OLESetData(Data As _
        VB.DataObject, DataFormat As Integer)
    If DataFormat = vbCFText Then
        Data.SetData txtSource.SelText, vbCfText
    End If
End Sub
```

95

Informing the Source When Data is Dropped

The *effect* argument of the OLEDragDrop event specifies how the data was incorporated into the target when the data was dropped. When this argument is set, the OLECompleteDrag event is triggered on the source with its *effect* argument set to this value. The source can then take the appropriate action: If a move is specified, the source deletes the data, for example.

The *effect* argument of the OLEDragDrop event uses the same constants as the *effect* argument of the OLEDragOver event to indicate the drop action. The following table lists these constants:

Constant	Value	Description
vbDropEffectNone	0	Drop target cannot accept the data.
vbDropEffectCopy	1	Drop results in a copy. The original data is untouched by the drag source.
VbDropEffectMove	2	Drag source removes the data.

96

The following example sets the *effect* argument to indicate the drop action.

```
Private Sub txtTarget_OLEDragDrop(Data As _
        VB.DataObject, Effect As Long, Button As _
        Integer, Shift As Integer, X As Single, _
        Y As Single)
    If Data.GetFormat(vbCFText) Then
        txtTarget.Text = Data.GetData(vbCFText)
    End If
    Effect = vbDropEffectMove
End Sub
```

97

On the source side, the OLECompleteDrag event is triggered when the source is dropped onto the target, or when the OLE drag-and-drop operation is canceled. OLECompleteDrag is the last event in the drag-and-drop operation.

The OLECompleteDrag event contains only one argument (*effect*), which is used to inform the source of the action that was taken when the data is dropped onto the target.

The *effect* argument returns the same values that are used by the *effect* argument of the other OLE drag-and-drop events: vbDropEffectNone, vbDropEffectCopy, and vbDropEffectMove.

By setting this argument after a move has been specified by the target and the source has been dropped into the target, for example, the source will delete the original data in the control. You should also use the OLECompleteDrag event to reset the mouse pointer if you specified a custom mouse pointer in the OLEGiveFeedback event. For example:

```
Private Sub txtSource_OLECompleteDrag(Effect As Long)
    If Effect = vbDropEffectMove Then
        txtSource.SelText = ""
    End If
    Screen.MousePointer = vbDefault
End Sub
```

98

Using the Mouse and Keyboard to Modify Drop Effects and User Feedback

You can enhance the OLEDragDrop and OLEDragOver events by using the *button* and *shift* arguments to respond to the state of the mouse buttons and the SHIFT, CTRL, and ALT keys. For instance, when dragging data into a control, you can allow the user to perform a copy operation by pressing the CTRL key, or a move operation by pressing the SHIFT key.

In the following example, the *shift* argument of the OLEDragDrop event is used to determine if the SHIFT key is pressed when the data is dropped. If it is, a move is performed. If it is not, a copy is performed.

```
Private Sub txtTarget_OLEDragDrop(Data As _
        VB.DataObject, Effect As Long, Button As _
        Integer, Shift As Integer, X As Single, _
        Y As Single)
    If Shift And vbCtrlMask Then
        txtTarget.Text = Data.GetData(vbCFText)
        Effect = vbDropEffectCopy
    Else
        txtTarget.Text = Data.GetData(vbCFText)
        Effect = vbDropEffectMove
    End If
End Sub
```

99

The *button* argument can be used to isolate and respond to the various mouse button states. For instance, you may want to allow the user to move the data by pressing both the right and left mouse buttons simultaneously.

To indicate to the user what action will be taken when the source object is dragged over the target when a mouse button or the SHIFT, CTRL, and ALT keys are pressed, you can set the *shift* and *button* arguments of the OLEDragOver event. For example, to inform the user what action will be taken when the SHIFT button is pressed during a drag operation, you can add the followinng code to the OLEDragOver event:

```
Private Sub txtTarget_OLEDragOver(Data As _
        VB.DataObject, Effect As Long, Button As _
```

```
        Integer, Shift As Integer, X As Single, _
        Y As Single, State As Integer)
    If Shift And vbCtrlMask Then
        Effect = vbDropEffectCopy
    Else
        Effect = vbDropEffectMove
    End If
End Sub
```

100

For More Information See "Detecting Mouse Buttons" and "Detecting SHIFT, CTRL, and ALT States" for more information on responding to mouse and keyboard states.

101

Creating a Custom Data Format

If the formats supplied in Visual Basic are insufficient for some specific purpose, you can create a custom data format for use in an OLE drag-and-drop operation. For example, a custom data format is useful if your application defines a unique data format that you need to drag between two instances of your application, or just within the application itself.

To create a custom data format, you have to call the Windows API RegisterClipboardFormat function. For example:

```
Private Declare Function RegisterClipboardFormat Lib _
        "user32.dll" Alias "RegisterClipboardFormatA" _
        (ByVal lpszFormat$) As Integer
Dim MyFormat As Integer
```

102

Once defined, you can use your custom format as you would any other DataObject object data format. For example:

```
Dim a() As Byte
a = Data.GetData(MyFormat)
```

103

To use this functionality, you have to place data into and retrieve data from the DataObject object as a Byte array. You can then assign your custom data format to a string variable because it is automatically converted.

Caution Retrieving your custom data format with the GetData method may yield unpredictable results.

Because Visual Basic doesn't understand your custom data format (because you defined it), it doesn't have a way to determine the size of the data. Visual Basic can determine the memory size of the Byte array because it has been allocated by Windows, but the operating system usually assigns more memory than is needed.

Therefore, when you retrieve a custom data format, you get back a Byte array containing at least, and possibly more than, the number of bytes that the source actually placed into the DataObject object. You must then correctly interpret your

custom data format when it is retrieved from the DataObject object. For example, in a simple string, you have to search for the NULL character and then truncate the string to that length.

Dragging Files From the Windows Explorer

You can use OLE drag-and-drop to drag files from the Windows Explorer into an appropriate Visual Basic control, or vice versa. For example, you can select a range of text files in the Windows Explorer and then open them all in a single text box control by dragging and dropping them onto the control.

To illustrate this, the following procedure uses a text box control and the OLEDragOver and OLEDragDrop events to open a range of text files using the Files property and the vbCFFiles data format of the DataObject object.

To drag text files into a text box control from the Windows Explorer

5 Start a new project in Visual Basic.

6 Add a text box control to the form. Set its OLEDropMode property to Manual. Set its MultiLine property to True and clear the Text property.

7 Add a function to select and index a range of files. For example:

```
1Sub DropFile(ByVal txt As TextBox, ByVal strFN$)
2    Dim iFile As Integer
3    iFile = FreeFile
         4
5    Open strFN For Input Access Read Lock Read _
6Write As #iFile
7    Dim Str$, strLine$
8    While Not EOF(iFile) And Len(Str) <= 32000
9        Line Input #iFile, strLine$
10       If Str <> "" Then Str = Str & vbCrLf
11       Str = Str & strLine
12   Wend
13   Close #iFile
                14
15   txt.SelStart = Len(txt)
16   txt.SelLength = 0
17   txt.SelText = Str
         18
19End Sub
```

104

8 Add the following procedure to the OLEDragOver event. The GetFormat method is used to test for a compatible data format (vbCFFiles).

```
20Private Sub Text1_OLEDragOver(Data As _
21VB.DataObject, Effect As Long, Button As Integer, _
22Shift As Integer, X As Single, Y As Single, State _
23As Integer)
24   If Data.GetFormat(vbCFFiles) Then
25       'If the data is in the proper format, _
26inform the source of the action to be taken
```

```
27  Effect = vbDropEffectCopy And Effect
28        Exit Sub
29  End If
30  'If the data is not desired format, no drop
31  Effect = vbDropEffectNone
              32
33End Sub
```

105

9 Finally, add the following procedure to the OLEDragDrop event.

```
34Private Sub Text1_OLEDragDrop(Data As _
35VB.DataObject, Effect As Long, Button As Integer, _
36Shift As Integer, X As Single, Y As Single)
37  If Data.GetFormat(vbCFFiles) Then
38        Dim vFN
39
40        For Each vFN In Data.Files
41              DropFile Text1, vFN
42        Next vFN
43  End If
44End Sub
```

106

10 Run the application, open the Windows Explorer, highlight several text files, and drag them into the text box control. Each of the text files will be opened in the text box.

10

Customizing the Mouse Pointer

You can use the MousePointer and MouseIcon properties to display a custom icon, cursor, or any one of a variety of predefined mouse pointers. Changing the mouse pointer gives you a way to inform the user that long background tasks are processing, that a control or window can be resized, or that a given control doesn't support drag-and-drop, for instance. Using custom icons or mouse pointers, you can express an endless range of visual information about the state and functionality of your application.

With the MousePointer property you can select any one of sixteen predefined pointers. These pointers represent various system events and procedures. The following table describes several of these pointers and their possible uses in your application.

Mouse pointer	Constant	Description
	vbHourglass	Alerts the user to changes in the state of the program. For example, displaying an hourglass tells the user to wait.
	vbSizePointer	Notifies the user of changes in function. For example, the double arrow sizing pointers tell users they can resize a window.
	vbNoDrop	Warns the user an action can't be performed. For example, the no drop pointer tells users they can't drop

a file at this location.

107

Each pointer option is represented by an integer value setting. The default setting is 0-Default and is usually displayed as the standard Windows arrow pointer. However, this setting is controlled by the operating system and can change if the system mouse settings have been changed by the user. To control the mouse pointer in your application, you set the MousePointer property to an appropriate value.

A complete list of mouse pointers is available by selecting the MousePointer property of a control or form and scanning the pull-down settings list or by using the Object Browser and searching for MousePointerConstants.

When you set the MousePointer property for a control, the pointer appears when the mouse is over the corresponding control. When you set the MousePointer property for a form, the selected pointer appears both when the mouse is over blank areas of the form and when the mouse is over controls with the MousePointer property set to 0-Default.

At run time you can set the value of the mouse pointer either by using the integer values or the Visual Basic mouse pointer constants. For example:

```
Form1.MousePointer = 11 'or vbHourglass
```

108

Icons and Cursors

You can set the mouse pointer to display a custom icon or cursor. Using custom icons or cursors allows you to further modify the look or functionality of your application. Icons are simply .ico files, like those shipped with Visual Basic. Cursors are .cur files and, like icons, are essentially bitmaps. Cursors, however, are created specifically to show the user where actions initiated by the mouse will take place — they can represent the state of the mouse and the current input location.

Cursors also contain hot spot information. The hot spot is a pixel which tracks the location of the cursor — the *x* and *y* coordinates. Typically, the hot spot is located at the center of the cursor. Icons, when loaded into Visual Basic through the MouseIcon property, are converted to the cursor format and the hot spot is set to the center pixel. The two differ in that the hot spot location of a .cur file can be changed, whereas that of an .ico file cannot. Cursor files can be edited in Image Editor, which is available in the Windows SDK.

Note Visual Basic does not support color cursors; they are displayed in black and white. Therefore, when using the MouseIcon property to create a custom mouse pointer, consider which is the more important need: color or location tracking. If color, use a color icon. If precise tracking is needed, use a cursor.

109

To use a custom icon or cursor, you set both the MousePointer and MouseIcon properties.

To use an .ico file as a mouse pointer

1. Select a form or control and set the MousePointer property to 99-Custom.
2. Load an .ico file into the MouseIcon property. For example, for a form:

```
45Form1.MouseIcon = LoadPicture("c:\Program _
46Files\Microsoft Visual _
47Basic\Icons\Computer\Disk04.ico")
```

11

Both properties must be set appropriately for an icon to appear as a mouse pointer. If no icon is loaded into MouseIcon when the MousePointer property is set to 99-Custom, the default mouse pointer is used. Likewise, if the MousePointer property is not set to 99-Custom, the setting of MouseIcon is ignored.

Note Visual Basic does not support animated cursor (.ani) files.

110

Responding to Keyboard Events

Keyboard events, along with mouse events, are the primary elements of a user's interaction with your program. Clicks and key presses trigger events and provide the means of data input and the basic forms of window and menu navigation.

Although the operating system provides the seamless back-end for all these actions, it's sometimes useful or necessary to modify or enhance them. The KeyPress, KeyUp, and KeyDown events allow you to make these modifications and enhancements.

Programming your application to respond to key events is referred to as writing a *keyboard handler*. A keyboard handler can work on two levels: at the control level and at the form level. The control level (*low-level*) handler allows you to program a specific control. For instance, you might want to convert all the typed text in a Textbox control to uppercase. A *form-level* handler allows the form to react to the key events first. The focus can then be shifted to a control or controls on the form, and the events can either be repeated or initiated.

With these key events you can write code to handle most of the keys on a standard keyboard. For information on dealing with international character sets and keyboards, see "International Issues."

Writing Low-Level Keyboard Handlers

Visual Basic provides three events that are recognized by forms and by any control that accepts keyboard input. They are described in the following table.

Keyboard event	Occurs
KeyPress	When a key corresponding to an ASCII character is pressed
KeyDown	As any key on the keyboard is pressed
KeyUp	As any key on the keyboard is released

111

Only the object that has the focus can receive a keyboard event. For keyboard events, a form has the focus only if it is active and no control on that form has the focus. This happens only on blank forms and forms on which all controls have been disabled. However, if you set the KeyPreview property on a form to True, the form receives all keyboard events for every control on the form before the control recognizes them. This is extremely useful when you want to perform the same action whenever a certain key is pressed, regardless of which control has the focus at the time.

The KeyDown and KeyUp events provide the lowest level of keyboard response. Use these events to detect a condition that the KeyPress event is unable to detect, for instance:

- Special combinations of SHIFT, CTRL, and ALT keys.
- Arrow keys. Note that some controls (command buttons, option buttons, and check boxes) do not receive arrow-key events: Instead, arrow keys cause movement to another control.
- PAGEUP and PAGEDOWN.
- Distinguishing the numeric keypad from numbers on the typewriter keys.
- Responding to a key being released as well as pressed (KeyPress responds only to a key being pressed).
- Function keys not attached to menu commands.

12

The keyboard events are not mutually exclusive. When the user presses a key, both the KeyDown and KeyPress events are generated, followed by a KeyUp event when the user releases the key. When the user presses one of the keys that KeyPress does not detect, only a KeyDown event occurs, followed by a KeyUp event.

Before using the KeyUp and KeyDown events, make sure that the KeyPress event isn't sufficient. This event detects keys that correspond to all the standard ASCII characters: letters, digits, and punctuation on a standard keyboard, as well as the ENTER, TAB, and BACKSPACE keys. It's generally easier to write code for the KeyPress event.

You also should consider using shortcut and access keys, which are described in "Menu Basics" in "Forms, Controls, and Menus." Shortcut keys must be attached to menu commands, but they can include function keys (including some function-key – shift-key combinations). You can assign shortcut keys without writing additional code.

Note The Windows ANSI (American National Standards Institute) character set corresponds to the 256 characters that include the standard Latin alphabet, publishing marks (such as copyright symbol, em dash, ellipsis), as well as many alternate and accented letters. These characters are represented by a unique 1-byte numeric value (0-255). ASCII (American Standard Code for Information Interchange) is essentially a subset (0-127) of the ANSI character set and represents the standard letters, digits, and

punctuation on a standard keyboard. The two character sets are often referred to interchangeably.

112

The KeyPress Event

The KeyPress event occurs when any key that corresponds to an ASCII character is pressed. The ASCII character set represents not only the letters, digits, and punctuation on a standard keyboard but also most of the control keys. The KeyPress event only recognizes the ENTER, TAB, and BACKSPACE keys, however. The other function, editing, and navigation keys can be detected by the KeyDown and KeyUp events.

Use the KeyPress event whenever you want to process the standard ASCII characters. For example, if you want to force all the characters in a text box to be uppercase, you can use this event to change the case of the keys as they are typed:

```
Private Sub Text1_KeyPress (KeyAscii As Integer)
    KeyAscii = Asc(UCase(Chr(KeyAscii)))
End Sub
```

113

The *keyascii* argument returns an integer value corresponding to an ASCII character code. The procedure above uses Chr to convert the ASCII character code into the corresponding character, UCase to make the character uppercase, and Asc to turn the result back into a character code.

Using the same ASCII character codes, you can test whether a key recognized by the KeyPress event is pressed. For instance, the following event procedure uses KeyPress to detect if the user is pressing the BACKSPACE key:

```
Private Sub Text1_KeyPress (KeyAscii As Integer)
    If KeyAscii = 8 Then MsgBox "You pressed the _
        BACKSPACE key."
End Sub
```

114

You can also use the Visual Basic key-code constants in place of the character codes. The BACKSPACE key in the example above has an ASCII value of 8. The constant value for the BACKSPACE key is vbKeyBack.

For More Information A complete list of key code constants with corresponding ASCII values can be found by using the Object Browser and searching for KeyCodeConstants.

115

You can also use the KeyPress event to alter the default behavior of certain keys. For example, pressing ENTER when there is no Default button on the form causes a beep. You can avoid this beep by intercepting the ENTER key (character code 13) in the KeyPress event.

```
Private Sub Text1_KeyPress (KeyAscii As Integer)
    If KeyAscii = 13 Then KeyAscii = 0
End Sub
```

116

The KeyDown and KeyUp Events

The KeyUp and KeyDown events report the exact physical state of the keyboard itself: A key is pressed down (KeyDown) and a key is released (KeyUp). In contrast, the KeyPress event does not report the state of the keyboard directly — it doesn't recognize the up or down state of the key, it simply supplies the character that the key represents.

A further example helps to illustrate the difference. When the user types uppercase "A," the KeyDown event gets the ASCII code for "A." The KeyDown event gets the same code when the user types lowercase "a." To determine whether the character pressed is uppercase or lowercase, these events use the *shift* argument. In contrast, the KeyPress event treats the uppercase and lowercase forms of a letter as two separate ASCII characters.

The KeyDown and KeyUp events return information on the character typed by providing the following two arguments.

Argument	Description
keycode	Indicates the physical key pressed. In this case, "A" and "a" are returned as the same key. They have the identical keycode value. But note that "1" on the typewriter keys and "1" on the numeric keypad are returned as different keys, even though they generate the same character.
shift	Indicates the state of the SHIFT, CTRL, and ALT keys. Only by examining this argument can you determine whether an uppercase or lowercase letter was typed.

117

The Keycode Argument

The *keycode* argument identifies a key by the ASCII value or by the key-code constant. Key codes for letter keys are the same as the ASCII codes of the uppercase character of the letter. So the keycode for both "A" and "a" is the value returned by Asc("A"). The following example uses the KeyDown event to determine if the "A" key has been pressed:

```
Private Sub Text1_KeyDown(KeyCode As Integer, _
        Shift As Integer)
    If KeyCode = vbKeyA Then MsgBox "You pressed _
        the A key."
End Sub
```

118

Pressing SHIFT + "A" or "A" without the SHIFT key displays the message box — that is, the argument is true in each case. To determine if the uppercase or lowercase form of the letter has been pressed you need to use the *shift* argument. See the topic, "The Shift Argument" later in this chapter.

Key codes for the number and punctuation keys are the same as the ASCII code of the number on the key. So the *keycode* for both "1" and "!" is the value returned by Asc("1"). Again, to test for the "!" character you need to use the *shift* argument.

The KeyDown and KeyUp events can recognize most of the control keys on a standard keyboard. This includes the function keys (F1-F16), the editing keys (HOME, PAGE UP, DELETE, etc), the navigation keys (RIGHT, LEFT, UP, and DOWN ARROW), and the keypad. These keys can be tested for by using either the key-code constant or the equivalent ASCII value. For example:

```
Private Sub Text1_KeyDown(KeyCode As Integer, _
        Shift As Integer)
    If KeyCode = vbKeyHome Then MsgBox "You _
        pressed the HOME key."
End Sub
```

119

For More Information A complete list of key code constants with corresponding ASCII can be found by using the Object Browser and searching for KeyCodeConstants.

120

The Shift Argument

The key events use the *shift* argument in the same way that the mouse events do — as integer and constant values that represent the SHIFT, CTRL, and ALT keys. You can use the *shift* argument with KeyDown and KeyUp events to distinguish between uppercase and lowercase characters, or to test for the various mouse states.

Building on the previous example, you can use the *shift* argument to determine whether the uppercase form of a letter is pressed.

```
Private Sub Text1_KeyDown(KeyCode As Integer, _
        Shift As Integer)
    If KeyCode = vbKeyA And Shift = 1 _
    Then MsgBox "You pressed the uppercase A key."
End Sub
```

121

Like the mouse events, the KeyUp and KeyDown events can detect the SHIFT, CTRL, and ALT individually or as combinations. The following example tests for specific shift-key states.

Open a new project and add the variable ShiftKey to the Declarations section of the form:

```
Dim ShiftKey as Integer
```

122

Add a Textbox control to the form and this procedure in the KeyDown event:

```
Private Sub Text1_KeyDown(KeyCode As Integer, _
        Shift As Integer)
    ShiftKey = Shift And 7
    Select Case ShiftKey
        Case 1 ' or vbShiftMask
            Print "You pressed the SHIFT key."
        Case 2 ' or vbCtrlMask
            Print "You pressed the CTRL key."
        Case 4 ' or vbAltMask
            Print "You pressed the ALT key."
```

```
        Case 3
            Print "You pressed both SHIFT and CTRL."
        Case 5
            Print "You pressed both SHIFT and ALT."
        Case 6
            Print "You pressed both CTRL and ALT."
        Case 7
            Print "You pressed SHIFT, CTRL, and ALT."
        End Select
End Sub
```

123

As long as the Textbox control has the focus, each key or combination of keys prints a corresponding message to the form when pressed.

For More Information See "Detecting SHIFT, CTRL, and ALT States" earlier in this chapter.

124

Writing Form-Level Keyboard Handlers

Each KeyDown and KeyUp event is attached to a specific object. To write a keyboard handler that applies to all objects on the form, set the KeyPreview property of the form to True. When the KeyPreview property is set to True, the form recognizes the KeyPress, KeyUp, and KeyDown events for all controls on the form before the controls themselves recognize the events. This makes it very easy to provide a common response to a particular keystroke.

You can set the KeyPreview property of the form to True in the Properties window or through code in the Form_Load procedure:

```
Private Sub Form_Load
    Form1.KeyPreview = True
End Sub
```

125

You can test for the various key states on a form by declaring a ShiftKey variable and using the Select Case statement. The following procedure will print the message to the form regardless of which control has the focus.

Open a new project and add the variable ShiftKey to the Declarations section of the form:

```
Dim ShiftKey as Integer
```

126

Add a Textbox and a CommandButton control to the form. Add the following procedure to the form's KeyDown event:

```
Private Sub Form_KeyDown(KeyCode As Integer, _
        Shift As Integer)
    ShiftKey = Shift And 7
    Select Case ShiftKey
        Case 1 ' or vbShiftMask
            Print "You pressed the SHIFT key."
        Case 2 ' or vbCtrlMask
            Print "You pressed the CTRL key."
```

```
        Case 4 ' or vbAltMask
            Print "You pressed the ALT key."
    End Select
End Sub
```

127

If you have defined a shortcut key for a menu control, the Click event for that menu control occurs automatically when the user types that key, and no key event occurs.

Similarly, if there is a command button on the form with the Default property set to True, the ENTER key causes the Click event for that command button to occur instead of a key event. If there is a command button with the Cancel property set to True, the ESC key causes the Click event for that command button to occur instead of a key event.

For example, if you add a Click event procedure to the CommandButton and then set either the Default or Cancel properties to True, pressing the RETURN or ESC keys will override the KeyDown event. This procedure closes the application:

```
Private Sub Command1_Click()
    End
End Sub
```

128

Notice that the TAB key moves the focus from control to control and does not cause a key event unless every control on the form is disabled or has TabStop set to False.

When the KeyPreview property of the form is set to True, the form recognizes the keyboard events before the controls, but the events still occur for the controls. To prevent this, you can set the *keyascii* or keycode arguments in the form key-event procedures to 0. For example, if there is no default button on the form, you can use the ENTER key to move the focus from control to control:

```
Private Sub Form_KeyPress (KeyAscii As Integer)
Dim NextTabIndex As Integer, i As Integer
    If KeyAscii = 13 Then
        If Screen.ActiveControl.TabIndex = _
        Count - 1 Then
            NextTabIndex = 0
        Else
            NextTabIndex = Screen.ActiveControl._
            TabIndex + 1
        End If
        For i = 0 To Count - 1
            If Me.Controls(i).TabIndex = _
            NextTabIndex Then
                Me.Controls(i).SetFocus
                Exit For
            End If
        Next i
        KeyAscii = 0
    End If
End Sub
```

129

Because this code sets *keyascii* to 0 when it is 13, the controls never recognize the ENTER key being pressed, and their key-event procedures are never called.

Interrupting Background Processing

Your application may utilize long background processing to accomplish certain tasks. If this is the case, it is helpful to provide the user with a way to either switch to another application or interrupt or cancel the background task. The Windows operating environment gives users the first option: switching to another application by using the ALT+TAB key combination, for instance. You can provide the other options by writing code that responds when a user either clicks a cancel button or presses the ESC key.

In considering how to implement this in your application, it's important to understand how tasks from various applications are handled by the operating system. Windows is a preemptively multitasking operating system, which means that idle processor time is efficiently shared among background tasks. These background tasks can originate from the application the user is working with, from another application, or perhaps from some system-controlled events. Priority is always given to the application that the user is working with, however. This ensures that the mouse and keyboard always respond immediately.

Background processing can be placed into two categories: constant and intermittent. An example of a constant task would be copying a file from a server. Periodically updating a value would be an example of an intermittent task. Both types of tasks can be interrupted or canceled by the user. However, because background processing is usually a complex matter, it is important to consider how these tasks are initiated in the first place. The topic "Allowing Users to Interrupt Tasks" later in this chapter describes these considerations and techniques.

Allowing Users to Interrupt Tasks

During long background tasks, your application cannot respond to user input. Therefore, you should provide the user with a way to interrupt or cancel the background processing by writing code for either the mouse or keyboard events. For example, when a long background task is running, you can display a dialog box that contains a Cancel button that the user can initiate by clicking the ENTER key (if the focus is on the Cancel button) or by clicking on it with the mouse.

Note You may also want to give the user a visual cue when a long task is processing. For example, you might show the user how the task is progressing (using a Label or Gauge control, for instance), or by changing the mouse pointer to an hourglass.

130

There are several techniques, but no one way, to write code to handle background processing. One way to allow users to interrupt a task is to display a Cancel button

and allow its Click event to be processed. You can do this by placing the code for your background task in a timer event, using the following guidelines.

- Use static variables for information that must persist between occurrences of the Timer event procedure.
- When the Timer event gets control, allow it to run slightly longer than the time you specified for the Interval property. This ensures that your background task will use every bit of processor time the system can give it. The next Timer event will simply wait in the message queue until the last one is done.
- Use a fairly large value — five to ten seconds — for the timer's Interval property, as this makes for more efficient processing. Preemptive multitasking prevents other applications from being blocked, and users are generally tolerant of a slight delay in canceling a long task.
- Use the Enabled property of the Timer as a flag to prevent the background task from being initiated when it is already running.

13

For More Information See "The Timer Control" in "Using Visual Basic's Standard Controls."

131

Using DoEvents

Although Timer events are the best tool for background processing, particularly for very long tasks, the DoEvents function provides a convenient way to allow a task to be canceled. For example, the following code shows a "Process" button that changes to a "Cancel" button when it is clicked. Clicking it again interrupts the task it is performing.

```
' The original caption for this button is "Process".
Private Sub Command1_Click()
    ' Static variables are shared by all instances
    ' of a procedure.
    Static blnProcessing As Boolean
    Dim lngCt As Long
    Dim intYieldCt As Integer
    Dim dblDummy As Double
    ' When the button is clicked, test whether it's
     'already processing.
    If blnProcessing Then
        ' If processing is in progress, cancel it.
        blnProcessing = False
    Else
        Command1.Caption = "Cancel"
        blnProcessing = True
        lngCt = 0
        ' Perform a million floating-point
        ' multiplications. After every
        ' thousand, check for cancellation.
        Do While blnProcessing And (lngCt < 1000000)
            For intYieldCt = 1 To 1000
```

—49

```
            lngCt = lngCt + 1
            dblDummy = lngCt * 3.14159
        Next intYieldCt
        ' The DoEvents statement allows other
        ' events to occur, including pressing this
        ' button a second time.
        DoEvents
    Loop
    blnProcessing = False
    Command1.Caption = "Process"
    MsgBox lngCt & " multiplications were performed"
  End If
End Sub
```

132

DoEvents switches control to the operating-environment kernel. Control returns to your application as soon as all other applications in the environment have had a chance to respond to pending events. This doesn't cause the current application to give up the focus, but it does enable background events to be processed.

The results of this yielding may not always be what you expect. For example, the following Click-event code waits until ten seconds after the button was clicked and then displays a message. If the button is clicked while it is already waiting, the clicks will be finished in reverse order.

```
Private Sub Command2_Click()
  Static intClick As Integer
  Dim intClickNumber As Integer
  Dim dblEndTime As Double
    ' Each time the button is clicked,
    ' give it a unique number.
  intClick = intClick + 1
  intClickNumber = intClick
    ' Wait for ten seconds.
  dblEndTime = Timer + 10#
  Do While dblEndTime > Timer
    ' Do nothing but allow other
    ' applications to process
    ' their events.
    DoEvents
  Loop
  MsgBox "Click " & intClickNumber & " is finished"
End Sub
```

133

You may want to prevent an event procedure that gives up control with DoEvents from being called again before DoEvents returns. Otherwise, the procedure might end up being called endlessly, until system resources are exhausted. You can prevent this from happening either by temporarily disabling the control or by setting a static "flag" variable, as in the earlier example.

Avoiding DoEvents When Using Global Data

It may be perfectly safe for a function to be called again while it has yielded control with DoEvents. For example, this procedure tests for prime numbers and uses DoEvents to periodically enable other applications to process events:

```
Function PrimeStatus (TestVal As Long) As Integer
    Dim Lim As Integer
    PrimeStatus = True
    Lim = Sqr(TestVal)
    For I = 2 To Lim
        If TestVal Mod I = 0 Then
            PrimeStatus = False
            Exit For
        End If
        If I Mod 200 = 0 Then DoEvents
    Next I
End Function
```

134

This code calls the DoEvents statement once every 200 iterations. This allows the PrimeStatus procedure to continue calculations as long as needed while the rest of the environment responds to events.

Consider what happens during a DoEvents call. Execution of application code is suspended while other forms and applications process events. One of these events might be a button click that launches the PrimeStatus procedure again.

This causes PrimeStatus to be re-entered, but since each occurrence of the function has space on the stack for its parameters and local variables, there is no conflict. Of course, if PrimeStatus gets called too many times, an Out of Stack Space error could occur.

The situation would be very different if PrimeStatus used or changed module-level variables or global data. In that case, executing another instance of PrimeStatus before DoEvents could return might result in the values of the module data or global data being different than they were before DoEvents was called. The results of PrimeStatus would then be unpredictable.