Visual Basic includes sophisticated text and graphics capabilities for use in your applications. If you think of text as a visual element, you can see that size, shape and color can be used to enhance the information presented. Just as a newspaper uses headlines, columns and bullets to break the words into bite-sized chunks, text properties can help you emphasize important concepts and interesting details.

Visual Basic also provides graphics capabilities allowing you great flexibility in design, including the addition of animation by displaying a sequence of images.

This chapter describes ways of placing and manipulating text and graphics. Details on formatting, fonts, color palettes, and printing are included. By combining these capabilities with good design concepts, you can optimize the attractiveness and ease of use of your applications.

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

Contents

- Working with Fonts
- Displaying Text on Forms and Picture Boxes
- Formatting Numbers, Dates, and Times
- Working with Selected Text
- Transferring Text and Graphics with the Clipboard Object
- Understanding the Coordinate System
- Using Graphical Controls
- Using Graphics Methods
- Working with Color
- Using the Picture Object
- Printing

2

Sample Applications: Blanker.vbp, Palettes.vbp

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

Working with Fonts

Text is displayed using a *font* — a set of characters of the same typeface, available in a particular size, style, and weight.

The Windows 95 and Windows NT operating systems provide you and your users with a complete set of standard fonts. TrueType fonts are scaleable, which means they can reproduce a character at any size. When you select a TrueType font, it is rendered into the selected point size and displayed as a bitmap on the screen.

When printing, the selected TrueType font or fonts are rendered into the appropriate size and then sent to the printer. Therefore, there is no need for separate screen and printer fonts. Printer fonts will be substituted for TrueType fonts, however, if an equivalent font is available, which increases print speed.

Choosing Fonts for Your Application

Remember that a user of your application may not have the fonts you used to create the application. If you select a TrueType font that a user doesn't have, Windows selects the closest matching font on the user's system. Depending on the design of your application, this may cause problems for the user. For example, the font Windows selects may enlarge text so that labels overlap on the screen.

One way to avoid font problems is to distribute the necessary fonts with your application. (You will probably need to obtain permission from the copyright holder of the font to distribute it with your application.)

You can also program your application to check among the fonts available in the operating system for the fonts you use. If the font doesn't reside in the operating system, you can program the application to choose a different font from the list.

Another way to avoid font problems is to use fonts users are most likely to have on their systems. If you use fonts from a specific version of Windows, you may have to specify that version as a system requirement of your application.

Checking Available Fonts

Your program can easily determine whether matching fonts are available on both the user's system and printer. The Fonts property applies to the Printer and Screen objects. An array returned by the Fonts property is a list of all of the fonts available to a printer or screen. You can iterate through the property array, and then search for matching name strings. This code example determines whether the system has a printer font that matches the font of the selected form:

```
Private Sub Form_Click ()
Dim I As Integer, Flag As Boolean
    For I = 0 To Printer.FontCount - 1
        Flag = StrComp (Font.Name,Printer.Fonts(I), 1)
        If Flag = True Then
            Debug.Print "There is a matching font."
            Exit For
        End If
    Next I
```

```
End Sub
```

3

For More Information For information about setting font properties, see "Setting Font Characteristics" later in this chapter. For information about fonts in East Asian systems, see "Font, Display, and Print Considerations in a DBCS Environment" in "International Issues."

4

Setting Font Characteristics

Forms, controls that display text (as text or captions), and the Printer object support a Font property, which determines the visual characteristics of text, including:

- Font name (typeface)
- Font size (in points)
- Special characteristics (bold, italic, underline, or strikethrough)

1

For details on the Printer object, see “Printing from an Application" later in this chapter.

Setting Font Properties

You can set any of the font properties at design time by double-clicking Font in the Properties window and setting the properties in the Font dialog box.

At run time, you set font characteristics by setting the Font object’s properties for each form and control. The following table describes the properties for the Font object.

Property	Type	Description
Name	String	Specifies name of font, such as Arial or Courier.
Size	Single	Specifies font size in points (72 points to an inch when printed).
Bold	Boolean	If True, the text is bold.
Italic	Boolean	If True, the text is italic.
StrikeThrough	Boolean	If True, Visual Basic strikes through the text.
Underline	Boolean	If True, the text is underlined.
Weight	Integer	Returns or sets the weight of the font. Above a certain weight, the Bold property is forced to True.

5

For example, the following statements set various font properties for a label named lblYearToDate:

```
With lblYearToDate.Font
     .Name = "Arial"           ' Change the font to Arial.
     .Bold = True              ' Make the font bold.
End With
```

6

The order in which you select font properties is important, because not all fonts support all font variations. Set the Name property first. Then you can set any of the Boolean properties, such as Bold and Italic, to True or False.

You can also store a set of font properties in a Font object. You can declare a Font object just as you would any other object, using the StdFont class:

```
Dim MyFont As New StdFont
With MyFont
      .Name = "Arial"
      .Size = 10
      .Bold = True
End With
```

7

Note   Before you can create a new Font object, you must use the References dialog box (available from the Project menu) to create a reference to Standard OLE Types.

8

You can then easily switch from one set of font properties to another, by setting the form or control's Font object to the new object:

```
Set lblYearToDate.Font = MyFont
```

9

Working with Small Fonts

When setting the Size property to sizes smaller than 8 points, Windows automatically changes to a different font if the selected font isn't supported in the smaller size. To avoid unpredictable results, first set the Size property when you use a font size smaller than 8 points. Set the Name property next. Then set the Size property again, followed by additional font properties.

Applying Font Properties to Specific Objects

The effect of setting font properties varies depending on the technique used to display text. If the text is specified by a property (such as Text or Caption), then changing a font property applies to all the text in that control. Labels, text boxes, frames, buttons, check boxes, and all the file-system controls use a property to specify text.

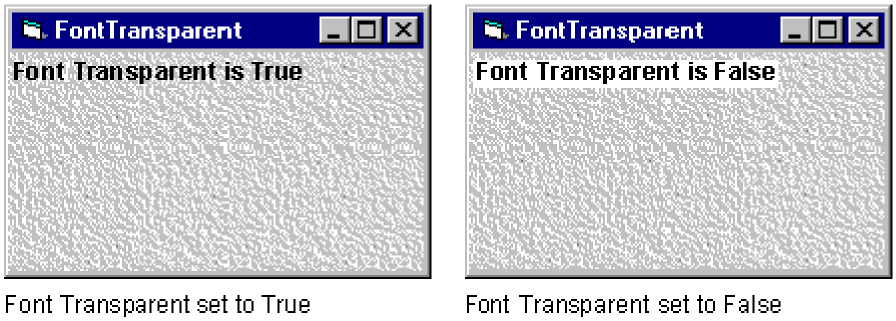
If the application shows text with the Print method, then changing a font property affects all uses of Print after the property change. Text printed before the property change is not affected. Only forms, picture boxes, and the Debug and Printer objects support the Print method.

Because changes in font properties apply to all the text in text boxes and labels, you cannot mix fonts in these controls. If you need to mix fonts (for example, making some words bold but leaving others in normal font), then create a picture box and use the Print method to display text. "Displaying Text on Forms and Picture Boxes" explains how to use the Print method.

The FontTransparent Property

Forms and picture boxes have an additional font property, FontTransparent. When FontTransparent is True, the background shows through any text displayed on the form or picture box. Figure 12.1 shows the effects of the FontTransparent property.

Figure 12.1 The effects of the FontTransparent property



2

Displaying Text on Forms and Picture Boxes

To display text on a form or picture box, use the Print method, preceded by the name of the form or picture box. To send output text to a printer, use the Print method on the Printer object.

Using the Print Method

The Print method syntax is:

[*object*.]**Print** [*outputlist*] [{ ; | , }] 10

The *object* argument is optional; if omitted, the Print method applies to the current form.

For example, the following statements print messages to:

- A form named MyForm:

  1MyForm.Print "This is a form."
  2

- A picture box named picMiniMsg: 3

  3picMiniMsg.Print "This is a picture box."
  4

- The current form: 4

  5Print "This is the current form."
  6

- The Printer object: 5

```
7Printer.Print "This text is going to the printer."
8
```

6

The *outputlist* argument is the text that appears on the form or picture box. Multiple items in the *outputlist* argument must be separated by commas or semicolons or both, as explained in “Displaying Different Items on a Single Line" later in this chapter.

Truncated Text

If the form or picture box is too small to display all the text, the text is cut off. Where the form or picture box cuts off the text depends on the coordinates of the location at which you began printing the text. You cannot scroll through a form or picture box.

Layering

When you print text to a form, the text appears in a layer *behind* any controls that have been placed on the form. So printing to a form usually works best on a form specifically created to hold the text. For more information about how text and graphics appear in layers on a form, see “Layering Graphics with AutoRedraw and ClipControls" later in this chapter

Displaying Different Items on a Single Line

The items you display or print can include property values, constants, and variables (either string or numeric). The Print method, discussed in "Displaying Text on Forms and Picture Boxes," prints the value of numeric items. Positive number values have a leading and a trailing space. Negative numeric values display their sign instead of a leading space.

Use a semicolon (;) or a comma (,) to separate one item from the next. If you use a semicolon, Visual Basic prints one item after another, without intervening spaces. If you use a comma, Visual Basic skips to the next tab column.

For example, the following statement prints to the current form:

```
Print "The value of X is "; X; "and the value of Y _
    is "; Y
```

11

If X contains the value 2 and Y contains the value 7, the statement produces this output:

```
The value of X is 2 and the value of Y is 7
```

12

By default, each Print method prints the text and moves to the next line. If there are no items, Print simply skips a line. A series of Print statements (in the following example, for a picture box named picLineCount) automatically uses separate lines:

```
picLineCount.Print "This is line 1."
picLineCount.Print "This is line 2."
```

13

By placing a semicolon (or comma) at the end of the first statement, however, you cause the output of the next Print statement to appear on the same line:

```
picLineCount.Print "This all appears ";
picLineCount.Print "on the same line."
```

14

Displaying Print Output at a Specific Location

You can control placement of Print output by specifying the drawing coordinates, using either or both of these techniques:

- Use the Cls (clear) method to erase a form or picture box and reset the drawing coordinates to the origin (0,0).
- Set drawing coordinates with the CurrentX and CurrentY properties.

7

The Cls Method

All the text and graphics on the object that were created with Print and graphics methods can be deleted with the Cls method. The Cls method also resets the drawing coordinates to the origin (0,0), which is the upper-left corner by default. For example, these statements clear:

- A picture box named Picture1:

```
9Picture1.Cls
```

8

- The current form:

```
10Cls
```

9

Setting Drawing Coordinates

You can set the drawing coordinates of forms and picture boxes directly with the CurrentX and CurrentY properties. For example, these statements reset the drawing coordinates to the upper-left corner for Picture1 and for the current form:

- A picture box named Picture1:

```
11Picture1.CurrentX = 0
12Picture1.CurrentY = 0
```

10

- The current form:

```
13CurrentX = 0
14CurrentY = 0
```

11

Any new text you print appears on top of any text and graphics already at that location. To erase text selectively, draw a box with the Line method and fill it with the background color. Keep in mind that the drawing coordinates specified by CurrentX and CurrentY usually change location when you use a graphics method.

By default, forms and picture boxes use a coordinate system where each unit corresponds to a twip (1,440 twips equal an inch, and approximately 567 twips equal a centimeter). You may want to change the ScaleMode property of the form, picture box, or Printer object from twips to points, because text height is measured in points.

Using the same unit of measure for the text and for the object where you will print the text makes it easier to calculate the position of the text.

For More Information For more information about twips and drawing coordinates, see "Understanding the Coordinate System" later in this chapter.

15

The TextHeight and TextWidth Methods

Before using the Print method, you can use the TextHeight and TextWidth methods to determine where to position the CurrentX and CurrentY properties. TextHeight returns the height of a line of text, taking into account the object's font size and style. The syntax is:

[*object*.]**TextHeight**(*string*)

16

If the *string* argument contains embedded carriage-return characters (Chr(13)), then the text corresponds to multiple lines, and TextHeight returns the height of the number of lines of text contained in the string. If there are no embedded carriage returns, TextHeight always returns the height of one line of text.

One way to use the TextHeight method is to set the CurrentY property to a particular line. For example, the following statements set the drawing coordinates to the beginning of the fifth line:

```
CurrentY = TextHeight("sample") * 4
CurrentX = 0
```

17

Assuming there are no carriage returns in the sample text, you would use this syntax to set CurrentY to the *nth* line:

CurrentY = [*object*.]**TextHeight**(*string*) * (*n* – 1)

18

If *object* is omitted, the method applies to the current form. The *object* argument can be a form, a picture box, or the Printer object.

The TextWidth method returns the width of a string, taking into account the object's font size and style. This method is useful because many fonts have proportional-width characters. The TextWidth method helps you determine whether the width of the string is larger than the width of the form, picture box, or Printer object.

For example, the following statements use TextWidth and TextHeight to center the text in a box by positioning CurrentX and CurrentY. The name of the box in this example is MealCard.

```
CurrentX = (BoxWidth - TextWidth("MealCard")) / 2
CurrentY = (Boxheight - TextHeight("MealCard")) / 2
```

19

Formatting Numbers, Dates, and Times

Visual Basic provides great flexibility in displaying number formats, as well as date and time formats. You can easily display international formats for numbers, dates, and times.

The Format function converts the numeric value to a text string and gives you control over the string's appearance. For example, you can specify the number of decimal places, leading or trailing zeros, and currency formats. The syntax is:

Format(*expression*[, *format*]

20

The *expression* argument specifies a number to convert, and the *format* argument is a string made up of symbols that shows how to format the number. The most commonly used symbols are listed in the following table.

Symbol	Description
0	Digit placeholder; prints a trailing or a leading zero in this position, if appropriate.
#	Digit placeholder; never prints trailing or leading zeros.
.	Decimal placeholder.
,	Thousands separator.
– + $ ( ) space	Literal character; characters are displayed exactly as typed into the format string.

21

Named Formats

Visual Basic provides several standard formats to use with the Format function. Instead of designating symbols in the *expression* argument, you specify these formats by name in the *format* argument of the Format function. Always enclose the format name in double quotation marks ("").

The following table lists the format names you can use.

Named format	Description
General Number	Shows numbers as entered.
Currency	Shows negative numbers inside parentheses.
Fixed	Shows at least one digit.
Standard	Uses a thousands separator.
Percent	Multiplies the value by 100 with a percent sign at the end.
Scientific	Uses standard scientific notation.
General Date	Shows date and time if *expression* contains both. If *expression* is only a date or a time, the missing information is not displayed.
Long Date	Uses the Long Date format specified in the Regional Settings dialog box of the Microsoft Windows Control Panel.

Medium Date	Uses the *dd-mmm-yy* format (for example, 03-Apr-93).
Short Date	Uses the Short Date format specified in the Regional Settings dialog box of the Windows Control Panel.
Long Time	Shows the hour, minute, second, and “AM” or “PM” using the *h:mm:ss* format.
Medium Time	Shows the hour, minute, and “AM” or “PM” using the “*hh:mm* AM/PM” format.
Short Time	Shows the hour and minute using the *hh:mm* format.
Yes/No	Any nonzero numeric value (usually – 1) is Yes. Zero is No.
True/False	Any nonzero numeric value (usually – 1) is True. Zero is False.
On/Off	Any nonzero numeric value (usually – 1) is On. Zero is Off.

22

The Format function supports many other special characters, such as the percentage placeholder and exponents.

Number Formats

The following number conversions assume that the country in the Windows Control Panel is set to “English (United States).”

Format syntax	**Result**
Format(8315.4, “00000.00”)	08315.40
Format(8315.4, “#####.##”)	8315.4
Format(8315.4, “##,##0.00”)	8,315.40
Format(315.4,“$##0.00”)	$315.40

23

The symbol for the decimal separator is a period (.), and the symbol for the thousands separator is a comma (,). However, the separator character that is actually displayed depends on the country specified in the Windows Control Panel.

Printing Formatted Dates and Times

To print formatted dates and times, use the Format function with symbols representing date and time. These examples use the Now and Format functions to identify and format the current date and time. The following examples assume that the Regional Settings dialog box of the Windows Control Panel is set to “English(United States)”.

Format syntax	**Result**
Format(Now, “*m/d/yy*”)	1/27/93
Format(Now, “*dddd, mmmm dd, yyyy*”)	Wednesday, January 27, 1993
Format(Now, “*d-mmm*”)	27-Jan
Format(Now, “*mmmm-yy*”)	January-93
Format(Now, “*hh:mm* AM/PM”)	07:18 AM
Format(Now, “*h:mm:ss* a/p”)	7:18:00 a

Format(Now, “*d-mmmm h:mm*”)	3-January 7:18

24

By using the Now function with the format “*ddddd*” and “*ttttt,*” you can print the current date and time in a format appropriate for the selection in the Regional Settings dialog box of the Windows Control Panel.

Country	Format syntax	Result
Sweden	Format(Now, “*ddddd ttttt*”)	1992-12-31 18.22.38
United Kingdom	Format(Now, “*ddddd ttttt*”)	31/12/92 18:22:38
Canada (French)	Format(Now, “*ddddd ttttt*”)	92-12-31 18:22:38
United States	Format(Now, “*ddddd ttttt*”)	12/31/92 6:22:38 PM

12

For More Information For more information about international considerations when using the Format function, see "Locale-Aware Functions" in “International Issues.” For more information about dates based on system locale, see "Writing International Code in Visual Basic" in “International Issues.”

25

Working with Selected Text

Text boxes and combo boxes have a series of properties for selected text that are especially useful when working with the Clipboard. These properties, which refer to the block of text selected (highlighted) inside the control, allow you to create cut-and-paste functions for the user. The following properties can all be changed at run time.

Property	Description
SelStart	A Long integer that specifies the starting position of the selected block of text. If no text is selected, this property specifies the position of the insertion point. A setting of 0 indicates the position just before the first character in the text box or combo box. A setting equal to the length of the text in the text box or combo box indicates the position just after the last character in the control.
SelLength	A Long integer that specifies the number of characters selected.
SelText	The String containing the selected characters (or an empty string, if no characters are selected).

26

You can control what text is selected by setting the SelStart and SelLength properties. For example, these statements highlight all the text in a text box:

```
Text1.SetFocus
' Start highlight before first character.
Text1.SelStart = 0
' Highlight to end of text.
Text1.SelLength = Len(Text1.Text)
```

27

If you assign a new string to SelText, that string replaces the selected text, and the insertion point is placed just after the end of the newly inserted text. For example, the following statement replaces the selected text with the string “I’ve just been inserted!”:

```
Text1.SelText = "I've just been inserted!"
```
28

If no text was selected, the string is simply pasted into the text box at the insertion point.

Transferring Text and Graphics with the Clipboard Object

The Clipboard object has no properties or events, but it has several methods that allow you to transfer data to and from the environment’s Clipboard. The Clipboard methods fall into three categories. The GetText and SetText methods are used to transfer text. The GetData and SetData methods transfer graphics. The GetFormat and Clear methods work with both text and graphic formats.

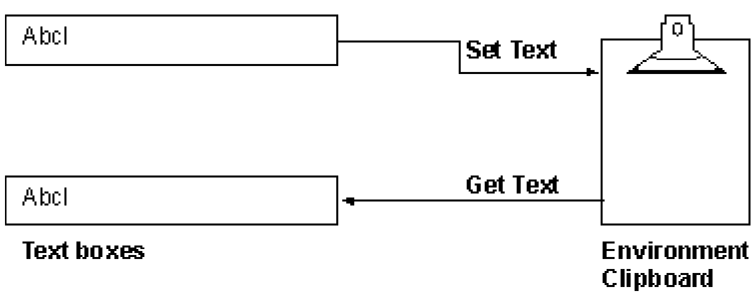
For More For information about transferring data within your application or between applications, see "OLE Drag and Drop" in "Responding to Mouse and Keyboard Events."

29

Cutting, Copying, and Pasting Text with the Clipboard

Two of the most useful Clipboard methods are SetText and GetText. These two methods transfer string data to and from the Clipboard, as shown in Figure 12.2.

Figure 12.2 Moving data to and from the Clipboard with SetText and GetText



13

SetText copies text onto the Clipboard, replacing whatever text was stored there before. You use SetText like a statement. Its syntax is:

Clipboard.SetText *data*[, *format*]

30

GetText returns text stored on the Clipboard. You use it like a function:

destination = **Clipboard.GetText**()

31

By combining the SetText and GetText methods with the selection properties introduced in "Working with Selected Text," you can easily write Copy, Cut, and Paste commands for a text box. The following event procedures implement these commands for controls named mnuCopy, mnuCut, and mnuPaste:

```
Private Sub mnuCopy_Click ()
    Clipboard.Clear
    Clipboard.SetText Text1.SelText
End Sub

Private Sub mnuCut_Click ()
    Clipboard.Clear
    Clipboard.SetText Text1.SelText
    Text1.SelText = ""
End Sub

Private Sub mnuPaste_Click ()
    Text1.SelText = Clipboard.GetText()
End Sub
```

32

Note The example works best if these are menu controls, because you can use menus while Text1 has the focus.

33

Notice that both the Copy and Cut procedures first empty the Clipboard with the Clear method. (The Clipboard is not cleared automatically because you may want to place data on the Clipboard in several different formats, as described in "Working with Multiple Formats on the Clipboard" later in this chapter.) Both the Copy and Cut procedures then copy the selected text in Text1 onto the Clipboard with the following statement:

```
Clipboard.SetText Text1.SelText
```

34

In the Paste command, the GetText method returns the string of text currently on the Clipboard. An assignment statement then copies this string into the selected portion of the text box (Text1.SelText). If no text is currently selected, Visual Basic places this text at the insertion point in the text box:

```
Text1.SelText = Clipboard.GetText()
```

35

This code assumes that all text is transferred to and from the text box Text1, but the user can copy, cut, and paste between Text1 and controls on other forms.

Because the Clipboard is shared by the entire environment, the user can also transfer text between Text1 and any application using the Clipboard.

Working with the ActiveControl Property

If you want the Copy, Cut, and Paste commands to work with any text box that has the focus, use the ActiveControl property of the Screen object. The following code provides a reference to whichever control has the focus:

```
Screen.ActiveControl
```

36

You can use this fragment just like any other reference to a control. If you know that the control is a text box, you can refer to any of the properties supported for text boxes, including Text, SelText, and SelLength. The following code assumes that the active control is a text box, and uses the SelText property:

```
Private Sub mnuCopy_Click ()
    Clipboard.Clear
    Clipboard.SetText Screen.ActiveControl.SelText
End Sub

Private Sub mnuCut_Click ()
    Clipboard.Clear
    Clipboard.SetText Screen.ActiveControl.SelText
    Screen.ActiveControl.SelText = ""
End Sub

Private Sub mnuPaste_Click ()
    Screen.ActiveControl.SelText = Clipboard.GetText()
End Sub
```

37

Working with Multiple Formats on the Clipboard

You can actually place several pieces of data on the Clipboard at the same time, as long as each piece is in a different format. This is useful because you don't know what application will be pasting the data, so supplying the data in several different formats enhances the chance that you will provide it in a format that the other application can use. The other Clipboard methods — GetData, SetData, and GetFormat — allow you to deal with data formats other than text by supplying a number that specifies the format. These formats are described in the following table, along with the corresponding number.

Constant	Description
vbCFLink	Dynamic data exchange link.
vbCFText	Text. Examples earlier in this chapter all use this format.
vbCFBitmap	Bitmap.
vbCFMetafile	Metafile.
vbCFDIB	Device-independent bitmap.
vbCFPalette	Color palette.

38

You can use the last four formats when cutting and pasting data from picture box controls. The following code provides generalized Cut, Copy, and Paste commands that work with any of the standard controls.

```
Private Sub mnuCopy_Click ()
    Clipboard.Clear
    If TypeOf Screen.ActiveControl Is TextBox Then
        Clipboard.SetText Screen.ActiveControl.SelText
    ElseIf TypeOf Screen.ActiveControl Is ComboBox Then
```

```
        Clipboard.SetText Screen.ActiveControl.Text
    ElseIf TypeOf Screen.ActiveControl Is PictureBox _
            Then
        Clipboard.SetData Screen.ActiveControl.Picture
    ElseIf TypeOf Screen.ActiveControl Is ListBox Then
        Clipboard.SetText Screen.ActiveControl.Text
    Else
        ' No action makes sense for the other controls.
    End If
End Sub

Private Sub mnuCut_Click ()
    ' First do the same as a copy.
    mnuCopy_Click
    ' Now clear contents of active control.
    If TypeOf Screen.ActiveControl Is TextBox Then
        Screen.ActiveControl.SelText = ""
    ElseIf TypeOf Screen.ActiveControl Is ComboBox Then
        Screen.ActiveControl.Text = ""
    ElseIf TypeOf Screen.ActiveControl Is PictureBox _
            Then
        Screen.ActiveControl.Picture = LoadPicture()
    ElseIf TypeOf Screen.ActiveControl Is ListBox Then
        Screen.ActiveControl.RemoveItem Screen.ActiveControl.ListIndex
    Else
        ' No action makes sense for the other controls.
    End If
End Sub

Private Sub mnuPaste_Click ()
    If TypeOf Screen.ActiveControl Is TextBox Then
        Screen.ActiveControl.SelText = Clipboard.GetText()
    ElseIf TypeOf Screen.ActiveControl Is ComboBox Then
        Screen.ActiveControl.Text = Clipboard.GetText()
    ElseIf TypeOf Screen.ActiveControl Is PictureBox _
            Then
        Screen.ActiveControl.Picture = _
            Clipboard.GetData()
    ElseIf TypeOf Screen.ActiveControl Is ListBox Then
        Screen.ActiveControl.AddItem Clipboard.GetText()
    Else
        ' No action makes sense for the other controls.
    End If
End Sub
```


Checking the Data Formats on the Clipboard

You can use the GetFormat method to determine whether the data on the Clipboard is in a particular format. For example, you can disable the Paste command depending on whether the data on the Clipboard is compatible with the currently active control.

```
Private Sub mnuEdit_Click ()
' Click event for the Edit menu.
    mnuCut.Enabled = True
```

```
    mnuCopy.Enabled = True
    mnuPaste.Enabled = False
    If TypeOf Screen.ActiveControl Is TextBox Then
        If Clipboard.GetFormat(vbCFText) Then mnuPaste.Enabled = True
    ElseIf TypeOf Screen.ActiveControl Is ComboBox Then
        If Clipboard.GetFormat(vbCFText) Then mnuPaste.Enabled = True
    ElseIf TypeOf Screen.ActiveControl Is ListBox Then
        If Clipboard.GetFormat(vbCFText) Then mnuPaste.Enabled = True
    ElseIf TypeOf Screen.ActiveControl Is PictureBox _
            Then
        If Clipboard.GetFormat(vbCFBitmap) Then mnuPaste.Enabled = True
    Else
        ' Can't cut or copy from the other types
        '     of controls.
        mnuCut.Enabled = False
        mnuCopy.Enabled = False
    End If
End Sub
```

40

Note You might also want to check for other data formats with the constants vbCFPalette, vbCFDIB, and vbCFMetafile. If you want to replace a picture's palette using Clipboard operations, you should request vbCFBitmap rather than vbCFDIB from the Clipboard. See "Working with 256 Colors" later in this chapter for more information on working with the color palette.

41

Understanding the Coordinate System

Every graphical operation described in this chapter (including resizing, moving, and drawing) uses the coordinate system of the drawing area or container. Although you can use the coordinate system to achieve graphical effects, it is also important to know how to use the coordinate system to define the location of forms and controls in your application.

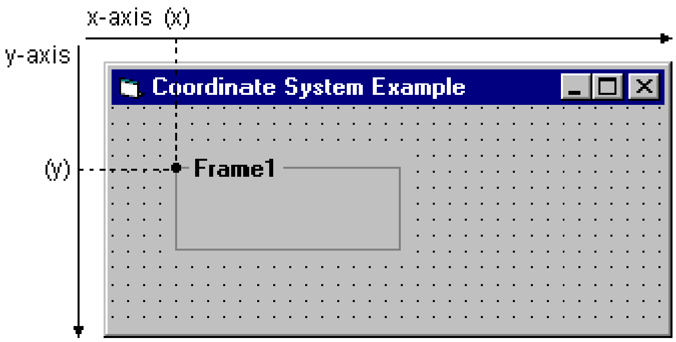
The coordinate system is a two-dimensional grid that defines locations on the screen, in a form, or other container (such as a picture box or Printer object). You define locations on this grid using coordinates in the form:

(*x*, *y*)

42

The value of *x* is the location of the point along the x-axis, with the default location of 0 at the extreme left. The value of y is the location of the point along the y-axis, with the default location of 0 at the extreme top. This coordinate system is illustrated in Figure 12.3.

Figure 12.3   The coordinate system of a form



The following rules apply to the Visual Basic coordinate system: 14

- When you move or resize a control, you use the coordinate system of the control's container. If you draw the object directly on the form, the form is the container. If you draw the control inside a frame or picture box, the frame or the control is the container.
- All graphics and Print methods use the coordinate system of the container. For example, statements that draw inside a picture box use the coordinate system of that control.
- Statements that resize or move a form always express the form's position and size in twips.

  1When you create code to resize or move a form, you should first check the Height and Width properties of the Screen object to make sure the form will fit on the screen.

- The upper-left corner of the screen is always (0, 0). The default coordinate system for any container starts with the (0, 0) coordinate in the upper-left corner of the container.

The units of measure used to define locations along these axes are collectively called the *scale*. In Visual Basic, each axis in the coordinate system can have its own scale. 15

You can change the direction of the axis, the starting point, and the scale of the coordinate system, but use the default system for now. "Changing an Object's Coordinate System" later in this chapter discusses how to make these changes.

Twips Explained

By default, all Visual Basic movement, sizing, and graphical-drawing statements use a unit of one twip. A *twip* is 1/20 of a printer's point (1,440 twips equal one inch, and 567 twips equal one centimeter). These measurements designate the size an object will be when printed. Actual physical distances on the screen vary according to the monitor size. “Changing an Object’s Coordinate System” describes how to select units other than twips.

Changing an Object's Coordinate System

You set the coordinate system for a particular object (form or control) using the object’s scale properties and the Scale method. You can use the coordinate system in one of three different ways:

- Use the default scale.
- Select one of several standard scales.
- Create a custom scale.

16

Changing the scale of the coordinate system can make it easier to size and position graphics on a form. For example, an application that creates bar charts in a picture box can change the coordinate system to divide the control into four columns, each representing a bar in the chart. The following sections explain how to set default, standard, and custom scales to change the coordinate system.

Using the Default Scale

Every form and picture box has several scale properties (ScaleLeft, ScaleTop, ScaleWidth, ScaleHeight, and ScaleMode) and one method (Scale) you can use to define the coordinate system. The default scale for objects in Visual Basic places the coordinate (0,0) at the upper-left corner of the object. The default scale uses twips.

If you want to return to the default scale, use the Scale method with no arguments.

Selecting a Standard Scale

Instead of defining units directly, you can define them in terms of a standard scale by setting the ScaleMode property to one of the settings shown in the following table.

ScaleMode setting	Description
0	User-defined. If you set ScaleWidth, ScaleHeight, ScaleTop, or ScaleLeft directly, the ScaleMode property is automatically set to 0.
1	Twips. This is the default scale. There are 1,440 twips to one inch.
2	Points. There are 72 points to one inch.
3	Pixels. A pixel is the smallest unit of resolution on the monitor or printer. The number of pixels per inch depends on the resolution of the device.
4	Characters. When printed, a character is 1/6 of an inch high and 1/12 of an inch wide.
5	Inches.
6	Millimeters.
7	Centimeters.

43

All of the modes in the table, except for 0 and 3, refer to printed lengths. For example, an item that is two units long when ScaleMode is set to 7 is two centimeters long when printed.

```
' Set scale to inches for this form.
ScaleMode = 5
' Set scale to pixels for picPicture1.
picPicture1.ScaleMode = 3
```

44

Setting a value for ScaleMode causes Visual Basic to redefine ScaleWidth and ScaleHeight so that they are consistent with the new scale. ScaleTop and ScaleLeft are then set to 0. Directly setting ScaleWidth, ScaleHeight, ScaleTop, or ScaleLeft automatically sets ScaleMode to 0.

Creating a Custom Scale

You can use an object's ScaleLeft, ScaleTop, ScaleWidth, and ScaleHeight properties to create a custom scale. Unlike the Scale method, these properties can be used either to set the scale or to get information about the current scale of the coordinate system.

Using ScaleLeft and ScaleTop

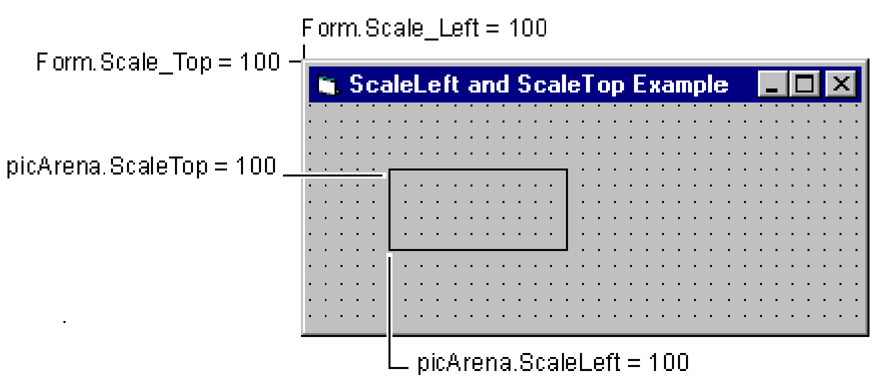
The ScaleLeft and ScaleTop properties assign numeric values to the upper-left corner of an object. For example, these statements set the value of the upper-left corner for the current form and upper-left corner for a picture box named picArena.

```
ScaleLeft = 100
ScaleTop = 100
picArena.ScaleLeft = 100
picArena.ScaleTop = 100
```

45

These scale values are shown in Figure 12.4.

Figure 12.4 The ScaleLeft and ScaleTop properties for a form and a control



17

These statements define the upper-left corner as (100, 100). Although the statements don't directly change the size or position of these objects, they alter the effect of subsequent statements. For example, a subsequent statement that sets a control's Top property to 100 places the object at the very top of its container.

Using ScaleWidth and ScaleHeight

The ScaleWidth and ScaleHeight properties define units in terms of the current width and height of the drawing area. For example:

```
ScaleWidth = 1000
ScaleHeight = 500
```

46

These statements define a horizontal unit as 1/1,000 of the current internal width of the form and a vertical unit as 1/500 of the current internal height of the form. If the form is later resized, the units remain the same.

Note ScaleWidth and ScaleHeight define units in terms of the internal dimensions of the object; these dimensions do not include the border thickness or the height of the menu or caption. Thus, ScaleWidth and ScaleHeight always refer to the amount of room available *inside* the object. The distinction between internal and external dimensions (specified by Width and Height) is particularly important with forms, which can have a thick border. The units can also differ: Width and Height are always expressed in terms of the *container's* coordinate system; ScaleWidth and ScaleHeight determine the coordinate system of the object itself.

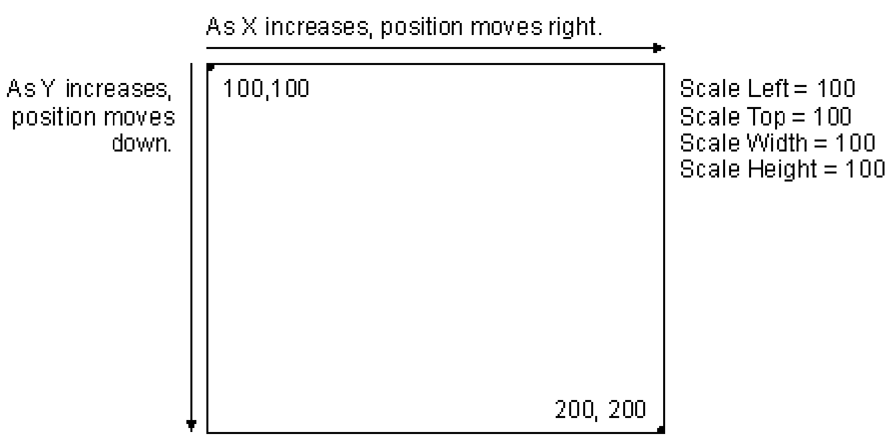
47

Setting Properties to Change the Coordinate System

All four of these scale properties can include fractions and they can also be negative numbers. Negative settings for the ScaleWidth and ScaleHeight properties change the orientation of the coordinate system.

The scale shown in Figure 12.5 has ScaleLeft, ScaleTop, ScaleWidth, and Scale Height all set to 100.

Figure 12.5 Scale running from (100, 100) to (200, 200)



18

Using the Scale Method to Change the Coordinate System

A more efficient way to change the coordinate system, other than setting individual properties, is to use the Scale method. You specify a custom scale using this syntax:

[*object*.]**Scale (***x1, y1***)** – **(***x2, y2***)** 48

The values of *x1* and *y1* determine the settings of the ScaleLeft and ScaleTop properties. The differences between the two x-coordinates and the two y-coordinates determine the settings of ScaleWidth and ScaleHeight, respectively. For example, suppose you set the coordinate system for a form by setting end points (100, 100) and (200, 200):

```
Scale (100, 100)-(200, 200)
```

49

This statement defines the form as 100 units wide and 100 units high. With this scale in place, the following statement moves a shape control one-fifth of the way across the form:

```
shpMover.Left = shpMover.Left + 20
```

50

Specifying a value of *x1 > x2* or *y1 > y2* has the same effect as setting ScaleWidth or ScaleHeight to a negative value.

Converting Scales

Use the ScaleX and ScaleY methods to convert from one scale mode to another scale mode. Those methods have the following syntax:

[*object*.]**ScaleX (***value* [, *fromScale* [, *toScale*]]
[*object*.]**ScaleY (***value* [, *fromScale*[,*toScale*]] 51

The destination *object* is a form, picture box, or Printer object. The *value* is expressed in the coordinate system specified by the scale mode *fromScale*. The value returned is expressed in the scale mode specified by *toScale*, or the scale mode of *object* if *toScale* is omitted. If *fromScale* is omitted, the scale mode for *value* is HIMETRIC.

HIMETRIC is the scale mode that specifies physical sizes. For example, the number of HIMETRIC units in a line of 10 centimeters is 10,000. The resulting line drawn on the screen is ten centimeters long, regardless of the size of the video display area. For information on the HIMETRIC scale mode and physical sizes, see the Microsoft Windows SDK.

The following statement stretches the content of the picture box control MyPic to twice its width. MyPic.Picture.Width returns the width of the picture contained in the picture control, which is a HIMETRIC value that needs to be converted into the scale mode of Form1.

```
Form1.PaintPicture MyPic.Picture, X, Y, _
    Form1.ScaleX(MyPic.Picture.Width) * 2
```

52

The following example illustrates two equivalent ways to specify a form’s Width to *np* pixels wide.

```
' The ScaleMode of the form is set to pixels.
ScaleMode = vbPixels

' Option 1:
' Temporarily set the form’s ScaleMode to twips.
ScaleMode = vbTwips
' ScaleX() returns the value in twips.
Width = Width - ScaleWidth + ScaleX(np, vbPixels)
' Set back the ScaleMode of the form to pixels.
ScaleMode = vbPixels
' Option 2:
' Conversion from pixels to twips without changing
'     the ScaleMode of the form.
Width = Width + ScaleX(np - ScaleWidth, vbPixels, _
     vbTwips)
```

53

Using Graphical Controls

Visual Basic provides three controls designed to create graphical effects in an application:

- The image control
- The line control
- The shape control

19

Advantages of Graphical Controls

The image, line, and shape controls are very useful for creating graphics at design time. One advantage of graphical controls is that they require fewer system resources than other Visual Basic controls, which improves the performance of your Visual Basic application.

Another advantage of graphical controls is that you can create graphics with less code than with graphics methods. For example, you can use either the Circle method or the shape control to place a circle on a form. The Circle method requires that you create the circle with code at run time, while you can simply draw the shape control on the form and set the appropriate properties at design time.

Limitations of Graphical Controls

While graphical controls are designed to maximize performance with minimal demands on the application, they accomplish this goal by limiting other features common to controls in Visual Basic. Graphical controls:

- Cannot appear on top of other controls, unless they are inside a container that can appear on top of other controls (such as a picture box).

- Cannot receive focus at run time.
- Cannot serve as containers for other controls.
- Do not have an hWnd property.

20

For More Information For information about the graphics methods, see "Using the Graphics Methods" later in this chapter. For information about the graphical controls, see "Using the Image Control," "Using the Line Control," and "Using the Shape Control" in "Using Visual Basic's Standard Controls."

54

Adding Pictures to Your Application

Pictures can be displayed in three places in Visual Basic applications:

- On a form
- In a picture box
- In an image control

21

Pictures can come from paint programs, such as those that ship with the various versions of Microsoft Windows, other graphics applications, or clip-art libraries. Visual Basic provides a large collection of icons you can use as graphics in applications. Visual Basic 5.0 allows you to add .jpeg and .gif files, as well as .bmp, .dib, .ico, .cur, .wmf, and .emf files to your applications. For more information about the graphics formats supported by Visual Basic, see "Using the Image Control" and "Using the Picture Box Control" in "Using Visual Basic's Standard Controls."

You use different techniques to add a picture to a form, a picture box, or an image control depending on whether you add the picture at design time or run time.

Adding a Picture at Design Time

There are two ways to add a picture at design time:

- Load a picture onto a form, or into a picture box or image control from a picture file:

  2In the Properties window, select Picture from the Properties list and click the Properties button. Visual Basic displays a dialog box, from which you select a picture file.

  3If you set the Picture property for a form, the picture you select is displayed on the form, behind any controls you've placed on it. Likewise, if you set the Picture property for a picture box, the picture is displayed in the box, behind any controls you've placed on it.

- Paste a picture onto a form or into a picture box or image control:

4Copy a picture from another application (such as Paintbrush) onto the Clipboard. Return to Visual Basic, select the form, picture box, or image control, and from the Edit menu, choose Paste.

22

Once you've set the Picture property for a form, picture box, or image control — either by loading or pasting a picture — the word displayed in the Settings box is "(Bitmap)," "(Icon)," or "(Metafile)." To change the setting, load or paste another picture. To set the Picture property to "(None)" again, double-click the word displayed in the Settings box and press the DEL key.

Adding a Picture at Run Time

There are four ways to add a picture at run time:

- Use the LoadPicture function to specify a file name and assign the picture to the Picture property.

5The following statement loads the file Cars.bmp into a picture box named picDisplay (you name a control by setting its Name property):

```
picDisplay.Picture = LoadPicture("C:\Picts\Cars.bmp")
```

23

6You can load a new picture file onto a form or into a picture box or image control whenever you want. Loading a new picture completely replaces the existing picture, although the source files of the pictures are never affected.

- Use the LoadResPicture function to assign a picture from the project's .res file into the Picture property.

7The following statement loads the bitmap resource ID, 10, from the resource file into a picture box named picResource:

```
Set picResource.Picture = LoadResPicture(10, _
   vbResBitmap)
```

24

- Copy a picture from one object to another.

8Once a picture is loaded or pasted onto a form or into a picture box or image control, you can assign it to other forms, picture boxes, or image controls at run time. For example, this statement copies a picture from a picture box named picDisplay to an image control named imgDisplay:

```
Set imgDisplay.Picture = picDisplay.Picture
```

25

- Copy a picture from the Clipboard object.

26

For More Information For more information about copying a picture from the Clipboard, see "Working with Multiple Formats on the Clipboard." For information on resource files, see "Working with Resource Files" in "More About Programming."

55

Note If you load or paste pictures from files at design time, the pictures are saved and loaded with the form, and the application copies pictures from one object to another. Then, when you create an .exe file, you don't need to give your users copies of the picture files; the .exe file itself contains the images.

Also, consider supplying a .res file and using LoadResPicture. The .res file gets built into the .exe, and the bitmaps are saved in a standard format that any resource editor can read. If you load pictures at run time with the LoadPicture function, you must supply the picture files to your users along with your application.

56

Removing a Picture at Run Time

You can also use the LoadPicture function to remove a picture at run time without replacing it with another picture. The following statement removes a picture from an image control named imgDisplay:

```
Set imgDisplay.Picture = LoadPicture("")
```

57

Moving and Sizing Pictures

If a form, picture box, or image control is moved (at design time or run time), its picture automatically moves with it. If a form, picture box, or image control is resized so that it is too small to display a picture, the picture gets clipped at the right and bottom. A picture also gets clipped if you load or copy it onto a form or into a picture box or image control that is too small to display all of it.

AutoSize Property

If you want a picture box to automatically expand to accommodate a new picture, set the AutoSize property for the picture box to True. Then when a picture is loaded or copied into the picture box at run time, Visual Basic automatically expands the control down and to the right enough to display all of the picture. If the image you load is larger than the edges of the form, it appears clipped because the form size doesn’t change.

You can also use the AutoSize property to automatically shrink a picture box to reflect the size of a new picture.

Note   Image controls do not have an AutoSize property, but automatically size themselves to fit the picture loaded into them. Forms don’t have an AutoSize property, and they do not automatically enlarge to display all of a picture.

58

Stretch Property of Image Controls

If you want a picture in an image control to automatically expand to fit a particular size, use the Stretch property. When the Stretch property is False, the image control automatically adjusts its size to fit the picture loaded into it. To resize the picture to fit the image control, set the Stretch property for the image control to True.

Selecting Art for the Picture Control

Where do you get picture files? If you want icons, you can use the Icon Library included with Visual Basic. You can find the icon files within the subdirectories of the main Visual Basic directory (\VB\Icons). You can create .bmp files with Microsoft PC Paintbrush, or you can buy a clip-art collection that includes bitmap or icon files, or metafiles. You can also create a resource (.res) file containing pictures.

For More Information See "Working with Resource Files" in "More About Programming, for more information on creating a resource file.

59

Introduction to Graphics Properties for Forms and Controls

Forms and various controls have graphics properties. The following table lists these properties.

Category	Properties
Display processing	AutoRedraw, ClipControls
Current drawing location	CurrentX, CurrentY
Drawing techniques	DrawMode, DrawStyle, DrawWidth, BorderStyle, BorderWidth
Filling techniques	FillColor, FillStyle
Colors	BackColor, ForeColor, BorderColor, FillColor

60

Forms and picture boxes have additional properties:

- Scale properties, as described in “Changing an Object’s Coordinate System” earlier in this chapter.
- Font properties, as described in “Setting Font Characteristics” earlier in this chapter.

27

There are two properties of forms and picture boxes you’ll probably want to use right away: BackColor and ForeColor. BackColor paints the background of the drawing area. If BackColor is light blue, then the entire area is light blue when you clear it. ForeColor (foreground) determines the color of text and graphics drawn on an object, although some graphics methods give you the option of using a different color. For more information about color, see "Working with Color" later in this chapter.

Creating Persistent Graphics with AutoRedraw

Each form and picture box has an AutoRedraw property. AutoRedraw is a Boolean property that, when set to True, causes graphics output to be saved in memory. You can use the AutoRedraw property to create persistent graphics.

Persistent Graphics

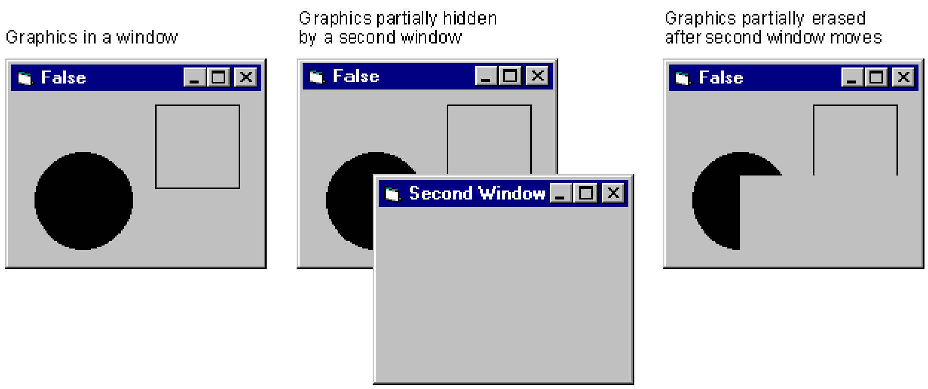
Microsoft Windows manipulates the screen image to create an illusion of overlapping windows. When one window is moved over another, temporarily hiding it, and is then moved away again, the window and its contents need to be redisplayed. Windows takes care of redisplaying the window and controls. But your Visual Basic application must handle redisplaying graphics in a form or picture box.

If you create graphics on the form using graphics methods, you usually want them to reappear exactly as you placed them (*persistent graphics*). You can use the AutoRedraw property to create persistent graphics.

AutoRedraw and Forms

The default setting of AutoRedraw is False. When AutoRedraw is set to False, any graphics created by graphics methods that appear on the form are lost if another window temporarily hides them. Also, graphics that extend beyond the edges of the form are lost if you enlarge the form. The effects of setting AutoRedraw to False are shown in Figure 12.6.

Figure 12.6 The effects of setting AutoRedraw to False

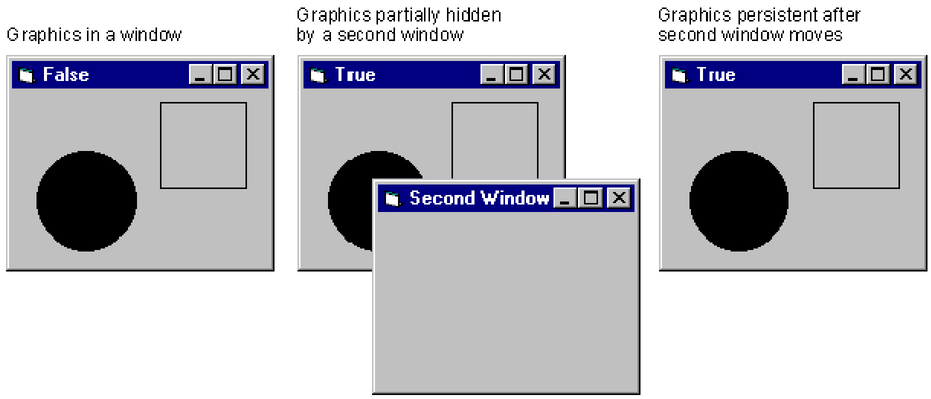


28

When the AutoRedraw property of a form is set to True, Visual Basic applies graphics methods to a “canvas” in memory. The application copies the contents of this memory canvas to redisplay graphics temporarily hidden by another window. In most cases, the size of this canvas for forms is the size of the screen. If the form’s MaxButton property is False and the border of the form is not sizable, the size of the canvas is the size of the form.

This canvas also lets the application save graphics that extend beyond the edges of the form when the form is resizable. The effects of setting AutoRedraw to True are shown in Figure 12.7.

Figure 12.7 The effects of setting AutoRedraw to True



AutoRedraw and Picture Boxes

When the AutoRedraw property of a picture box is set to True, Visual Basic saves only the visible contents of the picture box in memory. This is because the memory canvas used to save the contents of the picture box is the same size as the picture box. Graphics that extend outside the picture box are cropped and never appear later, even if the size of the picture box changes.

Using Nonpersistent Graphics

You can leave AutoRedraw set to False for the form and all its picture boxes to conserve memory. But then the graphics are not automatically persistent: You have to manage redrawing all graphics in code as needed.

You can include code in the Paint event for a form or picture box that redraws all lines, circles, and points as appropriate. This approach usually works best when you have a limited amount of graphics that you can reconstruct easily.

A Paint event procedure is called whenever part of a form or picture box needs to be redrawn — for example, when a window that covered the object moves away, or when resizing causes graphics to come back into view. If AutoRedraw is set to True, the object's Paint procedure is never called unless your application calls it explicitly. The visible contents of the object are stored in the memory canvas, so the Paint event isn't needed.

Keep in mind that the decision to use nonpersistent graphics can affect the way graphics paint on the form or container. "Clipping Regions with ClipControls" and "Layering Graphics with AutoRedraw and ClipControls" discuss other factors that may determine whether or not you should use nonpersistent graphics.

Changing AutoRedraw at Run Time

You can change the setting of AutoRedraw at run time. If AutoRedraw is False, graphics and output from the Print method are written only to the screen, not to memory. If you clear the object with the Cls method, any output written when AutoRedraw was set to True does not get cleared. This output is retained in memory, and you must set AutoRedraw to True again and then use the Cls method to clear it.

Clipping Regions with ClipControls

Each form, picture box, and frame control has a ClipControls property. ClipControls is a Boolean property that, when set to True, causes the container to define a clipping region when painting the container around all controls except:

- The shape control
- The line control
- The image control
- Labels
- Any custom graphical controls

30

By setting the ClipControls property to False, you can improve the speed with which a form paints to the screen. The speed improvement is greatest on forms with many controls that do not overlap, like dialog boxes.

Clipping Regions

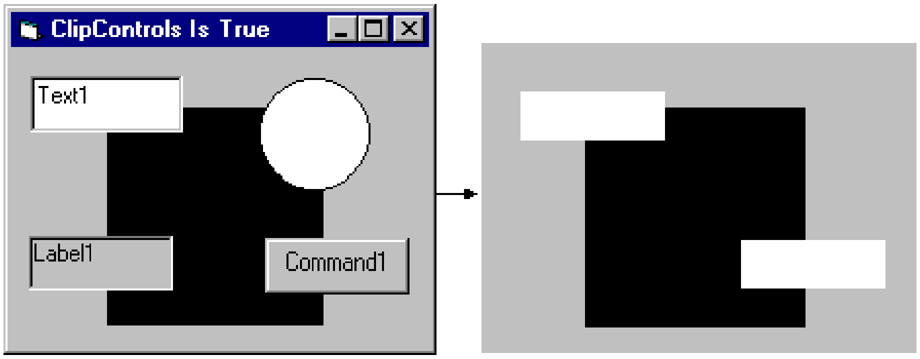
Clipping is the process of determining which parts of a form or container are painted when the form or container is displayed. The outline used to determine what parts of the form or container are painted or “clipped” defines the *clipping region* for that form or container. Clipping regions are useful when a Windows – based application needs to save one part of the display and simultaneously repaint the rest.

Clipping Forms and Containers

The default setting of ClipControls is True. When the ClipControls property is True, Windows defines a clipping region for the background of the form or container before a Paint event. This clipping region surrounds all nongraphical controls. When using ClipControls, labels act like graphical controls.

During a Paint event, Windows repaints only the background inside the clipping region, avoiding the nongraphical controls. Figure 12.8 shows a form with four controls, a box painted with the Line method, and the clipping region created for that form by setting ClipControls to True. Notice that the clipping region did not clip around the label or shape controls on the form. The box drawn in the background with the Line method paints only in the clipping region.

Figure 12.8 The clipping region created when ClipControls is True



Form with two nongraphical controls, a shape control, a label, and a box created with the Line method.

Clipping region shown in light gray; shape control and label not clipped. The box paints only in the clipping region.

31

When ClipControls is False, Windows does not define a clipping region for the background of the form or container before a Paint event. Also, output from graphics methods within the Paint event appears only in the parts of the form or container that need to be repainted. Since calculating and managing a clipping region takes time, setting ClipControls to False may cause forms with many nonoverlapping controls (such as complex dialog boxes) to display faster.

Note Avoid nesting controls with ClipControls set to True inside controls with ClipControls set to False. Doing so may result in the nested controls not repainting correctly. To fix this, set ClipControls to True for both the containers and the controls.

61

Layering Graphics with AutoRedraw and ClipControls

Different combinations of AutoRedraw and ClipControls have different effects on the way graphical controls and graphics methods paint to the screen.

As you create graphics, keep in mind that graphical controls and labels, nongraphical controls, and graphics methods appear on different layers in a container. The behavior of these layers depends on three factors:

- The AutoRedraw setting.
- The ClipControls setting.
- Whether graphics methods appear inside or outside the Paint event.

32

Normal Layering

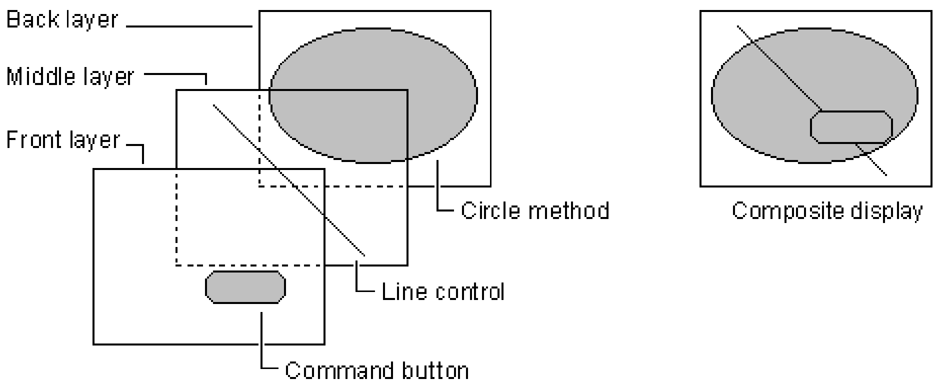
Usually, the layers of a form or other container are, from front to back, as follows:

Layer	Contents
Front	Nongraphical controls like command buttons, check boxes, and file controls.
Middle	Graphical controls and labels.
Back	Drawing space for the form or container. This is where the results of graphics methods appear.

62

Anything in one layer covers anything in the layer behind, so graphics you create with the graphical controls appear behind the other controls on the form, and all graphics you create with the graphics methods appear below all graphical and nongraphical controls. The normal arrangement of layers is shown in Figure 12.9.

Figure 12.9 Normal layering of graphics on a form



33

Effects on Layering

You can produce normal layering using any of several approaches. Combining settings for AutoRedraw and ClipControls and placing graphics methods inside or outside the Paint event affects layering and the performance of the application.

The following table lists the effects created by different combinations of AutoRedraw and ClipControls and placement of graphics methods.

AutoRedraw	ClipControls	Graphics methods in/out of Paint event	Layering behavior
True	True (default)	Paint event ignored	Normal layering.
True	False	Paint event ignored	Normal layering. Forms with many controls that do not overlap may paint faster because no clipping region is calculated or created.

False (default)	True (default)	In	Normal layering.
False	True	Out	Nongraphical controls in front. Graphics methods and graphical controls appear mixed in the middle and back layers. Not recommended.
False	False	In	Normal layering, affecting only pixels that were previously covered or that appear when resizing a form.
False	False	Out	Graphics methods and all controls appear mixed in the three layers. Not recommended.

63

The Effects of AutoRedraw

Setting AutoRedraw to True always produces normal layering. While using AutoRedraw is the easiest way to layer graphics, applications with large forms may suffer from reduced performance due to the memory demands of AutoRedraw.

The Effects of ClipControls

When AutoRedraw is True, the setting of ClipControls has no effect on how graphics layer on a form or in a container. But ClipControls can affect how fast the form displays. When ClipControls is False, the application doesn't create a clipping region. Not having to calculate or paint to avoid holes in a clipping region may cause the form to display faster.

Also, when AutoRedraw and ClipControls are both False, the application repaints only the pixels of a form or container that are exposed by:

- Covering the form or container with another window and then moving the window away.
- Resizing the form or container.

34

The Effects of the Paint Event

When AutoRedraw is False, the best place to use graphics methods is within the Paint event of the form or container. Confining graphics methods to the Paint event causes those methods to paint in a predictable sequence.

Using graphics methods outside a Paint event when AutoRedraw is False can produce unstable graphics. Each time the output of a graphics method appears on the form or container, it may cover any controls or graphics methods already there (if ClipControls is False). When an application uses more than a few graphics methods to create visual effects, managing the resulting output can be extremely difficult unless the methods are all confined to the Paint event.

Filename: in.doc Project: A Traditional User's Guide Template
Template: Author: Ruth Silverio Last Saved By: Christine
Revision #: 6 Page: 32 of 67 Printed: 10/02/96 03:42 PM

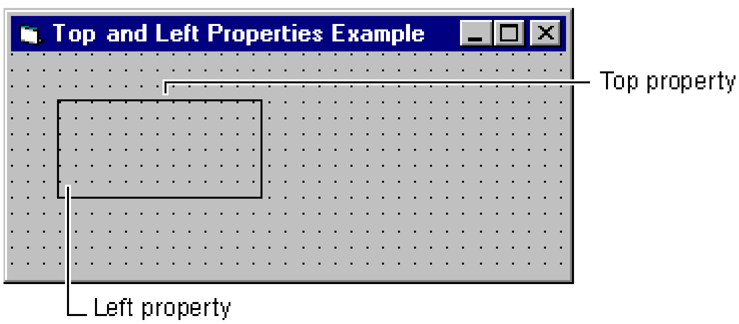
Moving Controls Dynamically

With Visual Basic, one of the easiest effects to achieve is moving a control at run time. You can either directly change the properties that define the position of a control or use the Move method.

Using the Left and Top Properties

The Left property is the distance between the upper-left corner of the control and the left side of the form. The Top property is the distance between the upper-left corner of the control and the top of the form. Figure 12.10 shows the Left and Top properties of a control.

Figure 12.10   The Left and Top properties



35

You can move a control by changing the settings of its Left and Top properties with statements such as these:

```
txtField1.Left = txtField1.Left + 200
txtField1.Top = txtField1.Top – 300
```

64

Moving a Line Control

As mentioned previously, line controls don’t have Left or Top properties. Instead, you use special properties to control the position of line controls on a form. The following table lists these properties and how they determine the position of a line control.

Property	Description
X1	The x-coordinate of the start of the line. The coordinate is given in current scale units. The start of the line is the end created when you start drawing.
Y1	The y-coordinate of the start of the line.
X2	The x-coordinate of the end of the line. The end of the line is the end created when you stop drawing.
Y2	The y-coordinate of the end of the line.

65

The Jumpy Line demo of the Blanker application randomly changes the position of a line control on the DemoForm using these statements:

```
' Set random X position for 1st line end.
linLineCtl.X1 = Int(DemoForm.Width * Rnd)
' Set random Y position for 1st line end.
linLineCtl.Y1 = Int(DemoForm.Height * Rnd)
' Set random X position for 2nd line end.
linLineCtl.X2 = Int(DemoForm.Width * Rnd)
' Set random Y position for 2nd line end.
linLineCtl.Y2 = Int(DemoForm.Height * Rnd)
' Clear stray pixels from moving line.
Cls
' Pause display briefly before next move.
Delay
```

66

Using the Move Method

Changing the Left and Top or X and Y properties produces a jerky effect as the control first moves horizontally and then vertically. The Move method produces a smoother diagonal movement.

The syntax for the Move method is:

[*object*.]**Move** *left* [, *top*[, *width*[, *height*]]] 67

The *object* is the form or control to be moved. If *object* is omitted, the current form moves. The *left* and *top* arguments are the new settings for the Left and Top properties of *object*, while *width* and *height* are new settings for its Width and Height properties. Only *left* is required, but to specify other arguments, you must include all arguments that appear in the argument list before the argument you want to specify.

Absolute Movement

Absolute movement occurs when you move an object to specific coordinates in its container. The following statement uses absolute movement to move a control named txtField1 to the coordinates (100, 200):

```
txtField1.Move 100, 200
```

68

Relative Movement

Relative movement occurs when you move an object by specifying the distance it should move from its current position. The following statement uses relative movement to move txtField1 to a position 100 twips down and to the right of its current position:

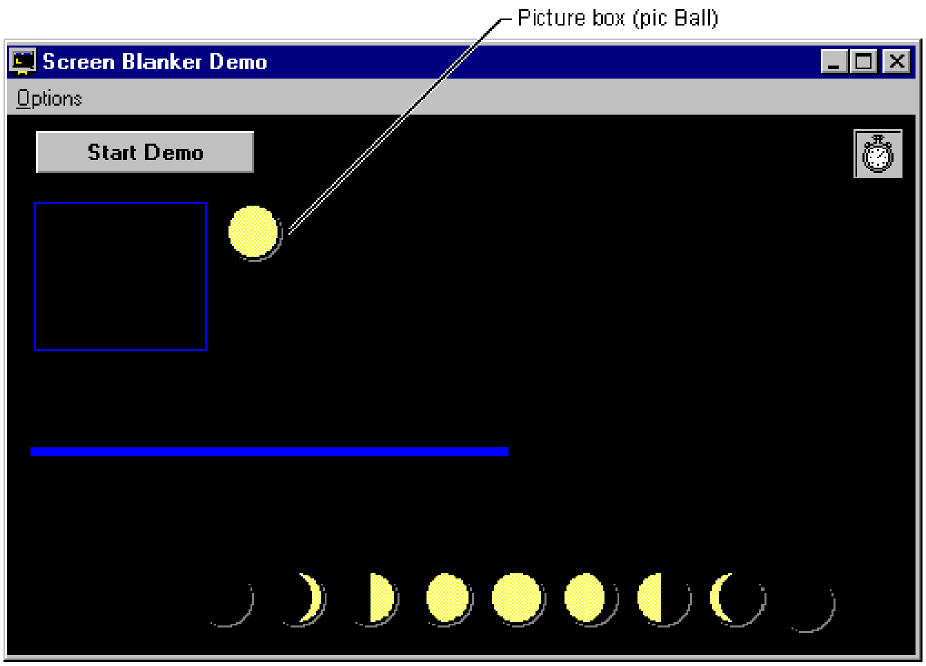
```
txtField1.Move txtField1.Left + 100, txtField1.Top _
    + 100
```

69

This section shows control movement in the Blanker sample application. The Rebound demo moves a picture box diagonally around the form, so the picture box appears to “bounce” off the sides of the form. This demo uses a picture box instead of an image control because the image control flickers as the movement causes it to repaint.

Figure 12.11 shows the main form of the Blanker application (DemoForm) and the picture box used in this example.

Figure 12.11   Picture box (picBall) in the Blanker application



36

The name of the picture box is picBall. This control begins moving around the form after you choose the Rebound command from the Options menu and then click the Start Demo button. The event procedure for this command button then calls the CtlMoveDemo procedure.

The CtlMoveDemo procedure randomly selects a starting direction from one of these four possibilities:

- Left and up
- Right and up
- Left and down
- Right and down

37

The picBall picture box moves along the chosen direction until the control reaches one of the four edges of the form. Then the picture box changes direction away from the edge it has reached; the variable Motion controls the direction. For example, when the picture box is moving left and up, this portion of the procedure changes the value of Motion and directs the code to move picBall in another direction.

The following statements come from the CtlMoveDemo procedure in the Blanker application:

```
Select Case Motion
Case 1
    ' If motion is left and up, move the control
    '     20 twips.
    picBall.Move picBall.Left - 20, picBall.Top - 20
    ' If control touches left edge, change motion
    '     to right and up.
    If picBall.Left <= 0 Then
        Motion = 2
    ' If control touches top edge, change motion to
    '     left and down.
    ElseIf picBall.Top <= 0 Then
        Motion = 4
    End If
```

70

Notice that the line of code that moves picBall subtracts 20 twips from the current values of its Left and Top properties to establish the new location of the control. This ensures that the control always moves relative to its current position.

The speed and smoothness of the control's movement depend on the number of twips (or other units) used in the Move method. Increasing the number of twips increases the speed but decreases the smoothness of motion. Decreasing the number of twips decreases the speed but improves the smoothness of the control's motion.

Resizing Controls Dynamically

In a Visual Basic application, you can change the size and shape of a picture box, image control, or form at run time, just as you can change its position.

The following properties affect size.

Property	Applies to	Description
Align	Picture boxes and Data controls	If set to align a picture box to the top (1) or bottom (2) of a form, the width of the picture box always equals the width of the inside of the form. If set to align a picture box to the left (3) or the right (4) of a form, the height of the picture box is the height of the inside of the form.
Height	All forms and all controls except timers, menus, and lines	Height of the object expressed in the scale mode of the form (twips by default).
Width	All forms and all controls except timers, menus, and	Width of the object expressed in the scale mode of the form (twips by default).

	lines	
AutoSize	Labels and picture boxes	If True, always causes Visual Basic to adjust the picture box dimensions to the size of the contents.
Stretch	Image controls	If True, the bitmap or metafile stretches to fit the size of the image control. If False, the size of the image control changes to match the size of the bitmap or metafile it contains.

71

In this example, a command button named cmdGrow grows larger each time the user clicks it:

```
Private Sub cmdGrow_Click ()
    cmdGrow.Height = cmdGrow.Height + 300
    cmdGrow.Width = cmdGrow.Width + 300
End Sub
```

72

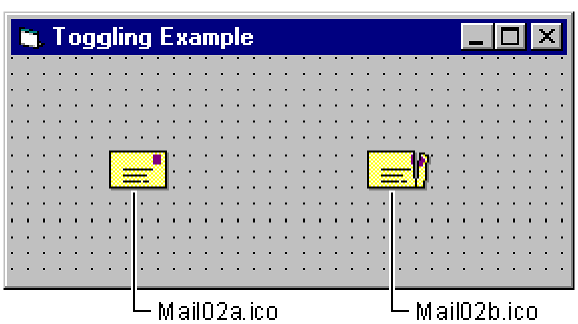
Creating Simple Animation

You can create simple animation by changing pictures at run time. The easiest way to do this is to toggle between two images. You can also use a series of pictures to create animation with several frames. Also, by moving the picture dynamically, you can create more elaborate effects.

Toggling Between Two Pictures

Some icons can be used in pairs. For instance, there are two matching envelope icons in the \Icon subdirectory, one with the envelope unopened and one with the envelope torn open, as shown in Figure 12.12. By switching, or *toggling,* between the two, you can create an animation that shows your user the status of mail.

Figure 12.12 Mail icons



38

The following statement changes the Picture property of an image control named imgMailStatus to toggle its picture from an unopened envelope to an open envelope.

```
imgMailStatus.Picture = imgMailOpen.Picture
```

73

Rotating Through Several Pictures

You can also rotate through several pictures to make longer animations. This technique is basically the same as toggling between two pictures, but requires the application to select which bitmap acts as the current image. One way to control the individual pictures in an animation is with a control array.

For More Information   See "Creating Arrays of Objects" in “Programming with Objects” for more information about control arrays.

74

The Blanker sample application includes an animation that shows a rotating moon. The Spinning Moon demo uses an array of nine image controls to create the animation. To view how the images in a control array work with each other at run time, choose Spinning Moon from the Options menu, and then choose the Start Demo button, which calls the ImageDemo procedure.

Using Graphics Methods

In addition to the graphical controls, Visual Basic provides several methods for creating graphics. The graphics methods, summarized in the following table, apply to forms and picture boxes.

Method	Description
Cls	Clears all graphics and Print output.
PSet	Sets the color of an individual pixel.
Point	Returns the color value of a specified point.
Line	Draws a line, rectangle, or filled-in box.
Circle	Draws a circle, ellipse, or arc.
PaintPicture	Paints graphics at arbitrary locations.

75

Note   The Print method can also be considered a graphics method, because its output is written to the object and is saved in the memory image (if AutoRedraw is on) just like the PSet, Line, and Circle methods. For more information about the Print method, see “Displaying Text on Forms and Picture Boxes” earlier in this chapter.

76

Advantages of Graphics Methods

The graphics methods work well in situations where using graphical controls require too much work. For example, creating gridlines on a graph would need an array of line controls but only a small amount of code using the Line method. Tracking the position of line controls in an array as the form changes size is more work than simply redrawing lines with the Line method.

When you want a visual effect to appear briefly on a form, such as a streak of color when you display an About dialog, you can write a couple of lines of code for this temporary effect instead of using another control.

Graphics methods offer some visual effects that are not available in the graphical controls. For example, you can only create arcs or paint individual pixels using the graphics methods. Graphics you create with these graphics methods appear on the form in a layer of their own. This layer is below all other controls on a form, so using the graphics methods can work well when you want to create graphics that appear behind everything else in your application.

For More Information See “Layering Graphics with AutoRedraw and ClipControls" earlier in this chapter.

77

Limitations of Graphics Methods

Creating graphics with the graphics methods takes place in code, which means you have to run the application to see the effect of a graphics method. Graphics methods therefore don't work as well as graphical controls for creating simple design elements of an interface. Changing the appearance of graphical controls at design time is easier than modifying and testing the code for a graphics method.

For More Information For information about creating graphical applications with the mouse events and the Line or Move methods, see "The MouseDown Event," "The MouseMove Event" and "Using Button to Enhance Graphical Mouse Applications" in "Responding to Mouse and Keyboard Events."

78

The Fundamentals of Drawing with Graphics Methods

Every graphics method draws output on a form, in a picture box, or to the Printer object. To indicate where you want to draw, precede a graphics method with the name of a form or picture box control. If you omit the object, Visual Basic assumes you want to draw on the form to which the code is attached. For example, the following statements draw a point on:

- A form named MyForm

```
19MyForm.PSet (500, 500)
20
```

39

- A picture box named picPicture1

```
21picPicture1.PSet (500, 500)
22
```

40

- The current form

```
23PSet (500, 500)
```

41

Each drawing area has its own coordinate system that determines what units apply to the coordinates. In addition, every drawing area has its own complete set of graphics properties.

For More Information See “Printing from an Application” later in this chapter for more information about the Printer object. See "Understanding the Coordinate System" for more information about coordinates.

79

Clearing the Drawing Area

Any time you want to clear a drawing area and start over, use the Cls method. The specified drawing area is repainted in the background color (BackColor):

[*object.*]**Cls**

80

Using the Cls method without a specified *object* clears the form to which the code is attached.

Plotting Points

Controlling an individual pixel is a simple graphics operation. The PSet method sets the color of a pixel at a specified point:

[*object.*]**PSet (***x*, *y***)**[, *color*]

81

The *x* and *y* arguments are single precision, so they can take either integer or fractional input. The input can be any numeric expression, including variables.

If you don’t include the *color* argument, PSet sets a pixel to the foreground color (ForeColor). For example, the following statements set various points on the current form (the form to which the code is attached), MyForm, and picPicture1:

```
PSet (300, 100)
PSet (10.75, 50.33)
MyForm.PSet (230, 1000)
picPicture1.PSet (1.5, 3.2)
```

82

Adding a *color* argument gives you more control:

```
' Set 50, 75 to bright blue.
PSet (50, 75), RGB(0, 0, 255)
```

83

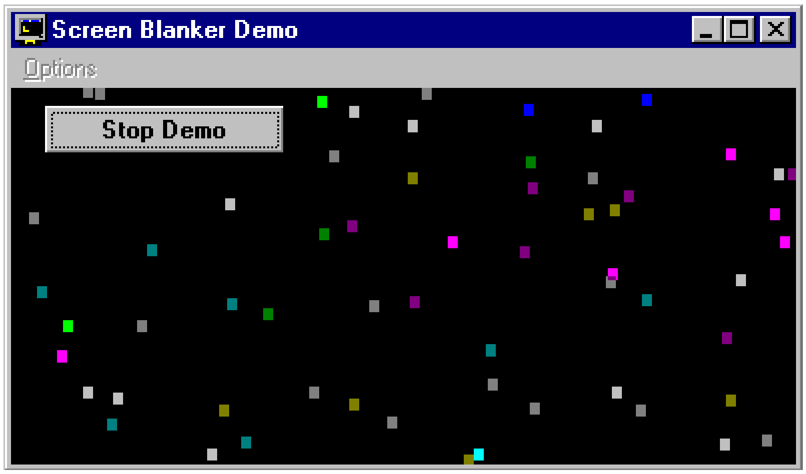
The Blanker application plots points with randomly selected colors to create the Confetti demo. The PSetDemo procedure creates the confetti:

```
Sub PSetDemo ()
    ' Set Red to random value.
    R = 255 * Rnd
    ' Set Green to random value.
    G = 255 * Rnd
    ' Set Blue to random value.
    B = 255 * Rnd
    ' Set horizontal position.
    XPos = Rnd * ScaleWidth
    ' Set vertical position.
    YPos = Rnd * ScaleHeight
    ' Plot point with random color.
    PSet (XPos, YPos), RGB(R, G, B)
```

```
End Sub
```

The resulting confetti display is shown in Figure 12.13. 84

Figure 12.13 Confetti display in the Blanker application



To “erase” a point, set it to the background color: 42

```
PSet (50, 75), BackColor
```

As described in "Drawing Lines and Shapes" later in this chapter, you can precede the (*x*, *y*) coordinates by Step, which makes the point relative to the last location drawn. 85

The Point method is closely related to the PSet method, but it returns the color value at a particular location:

```
PointColor = Point (500, 500)
```

Drawing Lines and Shapes 86

Although clearing the drawing area and plotting individual points can be useful, the most interesting graphics methods draw complete lines and shapes.

Drawing Lines

To draw a line between two coordinates, use the simple form of the Line method, which has this syntax:

[*object*.]**Line** [(*x1*, *y1*)]–(*x2*, *y2*)[, *color*]

Object is optional; if omitted, the method draws on the form to which the code is attached (the current form). The first pair of coordinates is also optional. As with all coordinate values, the *x* and *y* arguments can be either integer or fractional numbers. For example, this statement draws a slanted line on a form. 87

```
Line (500, 500)–(2000, 2000)
```

88

Visual Basic draws a line that includes the first end point, but not the last end point. This behavior is useful when drawing a closed figure from point to point. To draw the last point, use this syntax:

PSet Step(0, 0)[, *color*]

89

The first pair of coordinates (*x1, y1*) is optional. If you omit these coordinates, Visual Basic uses the object's current *x, y* location (drawing coordinates) as the end point. The current location can be specified with the CurrentX and CurrentY properties, but otherwise it is equal to the last point drawn by a previous graphics or Print method. If you haven't previously used a graphics or Print method or set CurrentX and CurrentY, the default location is the object's upper-left corner.

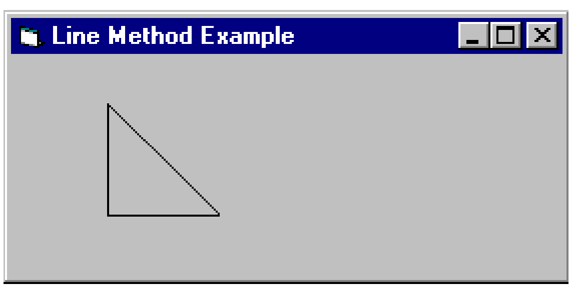
For example, the following statements draw a triangle by connecting three points.

```
' Set x-coordinate of starting point.
CurrentX = 1500
' Set y-coordinate of starting point.
CurrentY = 500
' Draw line down and right of starting point.
Line -(3000, 2000)
' Draw line to the left of current point.
Line -(1500, 2000)
' Draw line up and right to starting point.
Line -(1500, 500)
```

90

The results are shown in Figure 12.14.

Figure 12.14 A triangle drawn with the Line method



43

The Blanker application uses the Line method to create interesting patterns. To view this, from the Options menu, choose Crossfire, and then choose the Start Demo button.

The Step Keyword

The PSet, Line, and Circle methods specify one or more points using this syntax:

(*x*, *y*)

91

You can precede each of these points with the Step keyword, specifying that the location of the point is relative to the last point drawn. Visual Basic adds the *x* and *y* values to the values of the last point drawn. For example, the statement:

```
Line (100, 200)–(150, 250)
```

92

is equivalent to:

```
Line (100, 200)–Step(50, 50)
```

93

In many situations, the Step keyword saves you from having to constantly keep track of the last point drawn. Often you may be more interested in the relative position of two points than their absolute position.

Using the Color Argument

To vary the color of the line, use the optional *color* argument with graphics methods. For example, this statement draws a dark blue line:

```
Line (500, 500)–(2000, 2000), RGB(0, 0, 255)
```

94

If you omit the *color* argument, the ForeColor property for the object where the line is being drawn determines its color.

Drawing Boxes

You can draw and fill boxes using the Line method. The following example draws a box with an upper-left corner at (500, 500) and measuring 1,000 twips on each side:

```
Line (500, 500)–Step(1000, 0)
Line -Step(0, 1000)
Line -Step(–1000, 0)
Line -Step(0, –1000)
```

95

However, Visual Basic provides a much simpler way to draw a box. When you use the B option with the Line method, Visual Basic draws a rectangle, treating the specified points as opposite corners of the rectangle. Thus, you could replace the four statements of the previous example with the following:

```
Line (500, 500)–Step(1000, 1000), , B
```

96

Note that two commas are required before B, to indicate the color argument was skipped. The syntax of the Line method is covered in "Drawing Lines and Shapes" earlier in the chapter.

FillStyle and FillColor

As long as you do not change the setting of the FillStyle property, the box appears empty. (The box does get filled with the default FillStyle and settings, but FillStyle defaults to 1-Transparent.) You can change the FillStyle property to any of the settings listed in the following table.

Setting	Description
0	Solid. Fills in box with the color set for the FillColor property.
1	Transparent (the default). Graphical object appears empty, no matter what color is used.
2	Horizontal lines.
3	Vertical lines.
4	Upward diagonal lines.
5	Downward diagonal lines.
6	Crosshatch.
7	Diagonal crosshatch.

97

Thus, setting FillStyle to 0 fills the box solidly with the color set for the FillColor property.

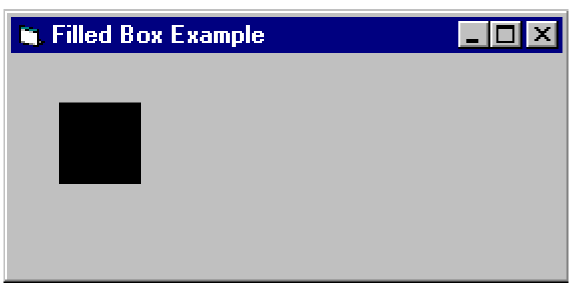
Another way to fill the box is to specify **F** after the **B**. (Note that **F** cannot be used without **B.**) When you use the **F** option, the Line method ignores FillColor and FillStyle. The box is always filled solid when you use the **F** option. The following statement fills the box with a solid pattern, using the ForeColor property:

```
Line (500, 500)–Step(1000, 1000), , BF
```

98

The result is shown in Figure 12.15.

Figure 12.15 A box filled with a solid pattern



44

Drawing Circles

The Circle method draws a variety of circular and elliptical (oval) shapes. In addition, Circle draws arcs (segments of circles) and pie-shaped wedges. You can produce many kinds of curved lines using variations of the Circle method.

To draw a circle, Visual Basic needs the location of a circle's center and the length of its radius. The syntax for a perfect circle is:

[*object*.]**Circle** [**Step**](*x, y*), *radius*[, *color*]

99

The brackets indicate that both *object* and the Step keyword are optional. If you don't specify *object*, the current form is assumed. The *x* and *y* arguments are the coordinates of the center, and *radius* is the radius of the circle. For example, this statement draws a circle with a center at (1200, 1000) and radius of 750:

```
Circle (1200, 1000), 750
```

100

The exact effect of this statement depends on the size and coordinate system of the form. Because the size of the form is unknown, you don't know if the circle will be visible. Using the drawing area's scale properties puts the center of the circle at the center of the form:

```
Circle ((ScaleWidth + ScaleLeft) / 2, (ScaleHeight + _
    ScaleTop) / 2), ScaleWidth / 4
```

101

For now, all you need to know about ScaleWidth and ScaleHeight is that they help position graphics in the center of a form.

For More Information "Changing an Object's Coordinate System" earlier in this chapter discusses the ScaleWidth and ScaleHeight properties in detail.

102

Note The radius of the circle is always specified in terms of horizontal units. If your coordinate system uses the same horizontal and vertical units (which it does by default), you can ignore this fact. However, if you use a custom scale, horizontal and vertical units may correspond to different distances. In the preceding examples, the radius is specified in horizontal units, and the actual height of the circle is guaranteed to be equal to its actual width.

103

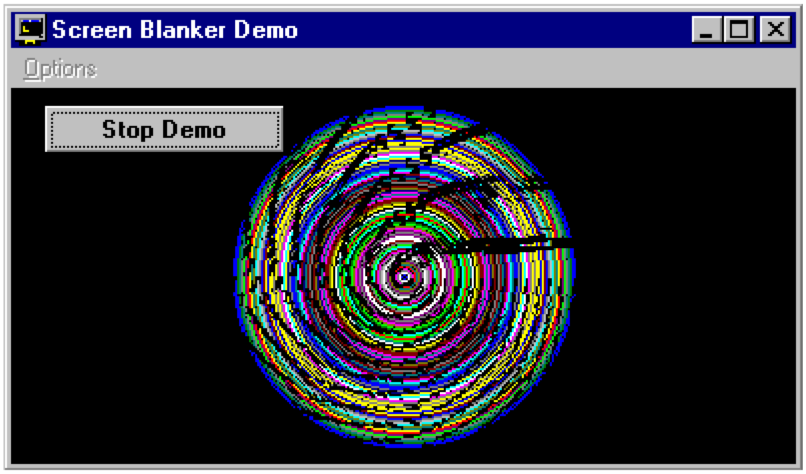
The Blanker application creates circles as part of the Rainbow Rug demo. This demo draws a series of dashed line circles around the center of the form. In time the circles resemble a woven circular rug. The CircleDemo procedure creates the circles in the Rainbow Rug demo with the following statements:

```
Sub CircleDemo ()
    Dim Radius
' Set Red to a random value.
    R = 255 * Rnd
' Set Green to a random value.
    G = 255 * Rnd
' Set Blue to a random value.
    B = 255 * Rnd
' Set x-coordinate in middle of form.
    XPos = ScaleWidth / 2
' Set y-coordinate in middle of form.
    YPos = ScaleHeight / 2
    ' Set radius between 0 & 50% of form height.
    Radius = ((YPos * 0.9) + 1) * Rnd
    ' Draw the circle using a random color.
    Circle (XPos, YPos), Radius, RGB(R, G, B)
End Sub
```

104

The results of the Rainbow Rug demo are shown in Figure 12.16.

Figure 12.16 The Rainbow Rug demo in the Blanker application



45

Drawing Arcs

To draw arcs with the Circle method, you need to give angle arguments in radians to define the *start* and the *end* of the arc. The syntax for drawing an arc is:

[*object*.]**Circle** [**Step**](*x*, *y*), *radius*, [*color*], *start*, *end*[, *aspect*] 105

If the *start* or *end* argument is negative, Visual Basic draws a line connecting the center of the circle to the negative end point. For example, the following procedure draws a pie with a slice removed.

```
Private Sub Form_Click ()
    Const PI = 3.14159265
    Circle (3500, 1500), 1000, , –PI / 2, –PI / 3
End Sub
```
106

Note The formula for converting from degrees to radians is to multiply degrees by Pi/180. 107

Drawing Ellipses

The aspect ratio of a circle controls whether or not it appears perfectly round (a circle) or elongated (an ellipse). The complete syntax for the Circle method is:

[*object*.]**Circle** [**Step**]**(***x, y***)**, *radius*, [*color*], [*start*], [*end*] [, *aspect*] 108

The *start* and *end* arguments are optional, but the commas are necessary if you want to skip arguments. For example, if you include the *radius* and *aspect* arguments, but no *color*, *start*, or *end* argument, you must add four successive commas to indicate that you're skipping the three arguments:

```
Circle (1000, 1000), 500, , , , 2
```
109

The *aspect* argument specifies the ratio of the vertical to horizontal dimensions. Here, *aspect* is a positive floating-point number. This means you can specify integer or fractional expressions, but not negative values. Large values for *aspect* produce ellipses stretched out along the vertical axis, while small values for *aspect* produce ellipses stretched out along the horizontal axis. Since an ellipse has two radii — one horizontal x-radius and one vertical y-radius — Visual Basic applies the single argument *radius* in a Circle statement to the longer axis. If *aspect* is less than one, *radius* is the x-radius; if *aspect* is greater than or equal to one, *radius* is the y-radius.

Note The *aspect* argument always specifies the ratio between the vertical and horizontal dimensions in terms of true physical distance. To ensure that this happens (even when you use a custom scale), the radius is specified in terms of horizontal units.

110

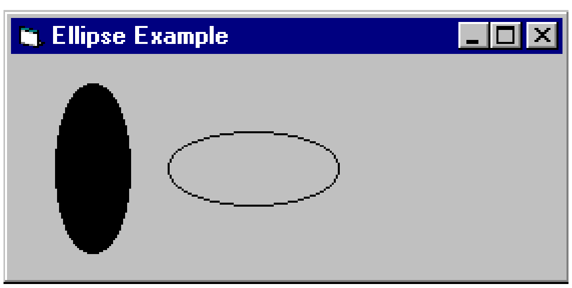
The following procedure illustrates how different *aspect* values determine whether Circle uses the *radius* argument as the x-radius or the y-radius of an ellipse:

```
Private Sub Form_Click ()
' Draw solid ellipse.
    FillStyle = 0
    Circle (600, 1000), 800, , , , 3
' Draw empty ellipse.
    FillStyle = 1
    Circle (1800, 1000), 800, , , , 1 / 3
End Sub
```

111

The output is shown in Figure 12.17.

Figure 12.17 Ellipses drawn with the Circle method



46

For More Information For more information about drawing circles and arcs, see "Drawing Circles" earlier in this chapter.

112

Painting Graphics at Arbitrary Locations

You can paint graphics at arbitrary locations on a form, on a picture box, and to the Printer object using the PaintPicture method. The syntax for the PaintPicture method is:

[*object*.]**PaintPicture** *pic*, *destX*, *destY*[, *destWidth*[, *destHeight*[, *srcX* _
[, *srcY*[, *srcWidth*[, *srcHeight*[, *Op*]]]]]]] 113

The destination *object* is the form, picture box, or Printer object where the *pic* picture is rendered. If *object* is omitted, the current form is assumed. The *pic* argument must be a Picture object, as from the Picture property of a form or control.

The *destX* and *destY* arguments are the horizontal and vertical locations where the picture will be rendered in the ScaleMode of *object*. The *destWidth* and *destHeight* arguments are optional and set the width and height with which the picture will be rendered in the destination *object*.

The *srcX* and *srcY* arguments are optional and define the x-coordinate and y-coordinate of the upper-left corner of a clipping region within *pic*.

The optional *Op* argument defines a raster operation (such as AND or XOR) that is performed on the picture as it is being painted on the destination *object*.

The PaintPicture method can be used in place of the BitBlt Windows API function to perform a wide variety of bit operations while moving a rectangular block of graphics from one position to any other position.

For example, you can use the PaintPicture method to create multiple copies of the same bitmap, and tile them on a form. Using this method is faster than moving picture controls on a form. The following code tiles 100 copies of a picture control and flips every picture horizontally by supplying a negative value for *destWidth.*

```
For i = 0 To 10
    For j = 0 To 10
        Form1.PaintPicture picF.Picture, j * _
            picF.Width, i * picF.Height, _
            picF.Width, -picF.Height
Next j, i
```

114

Specifying Line Width

The DrawWidth property specifies the width of the line for output from the graphics methods. The BorderWidth property specifies the outline thickness of line and shape controls.

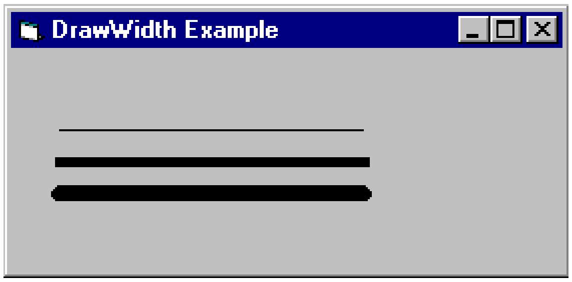
The following procedure draws lines of several different widths.

```
Private Sub Form_Click ()
    DrawWidth = 1
    Line (100, 1000)–(3000, 1000)
    DrawWidth = 5
    Line (100, 1500)–(3000, 1500)
    DrawWidth = 8
    Line (100, 2000)–(3000, 2000)
End Sub
```

115

The results are shown in Figure 12.18.

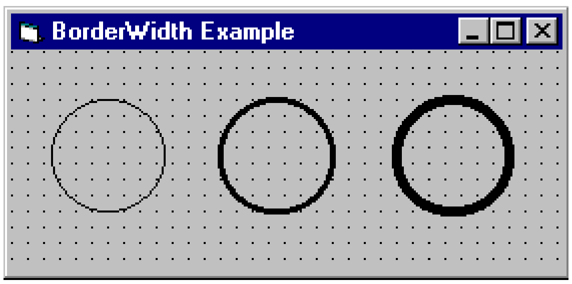
Figure 12.18 The effects of changing the DrawWidth property



47

Figure 12.19 shows three shape controls with different BorderWidth values.

Figure 12.19 The effects of changing the BorderWidth property



48

Specifying Solid or Broken Lines

The DrawStyle property specifies whether the lines created with graphics methods are solid or have a broken pattern. The BorderStyle property of a shape control serves the same function as the DrawStyle property, but applies to a variety of objects.

Note The BorderStyle property of a shape control serves a different purpose and uses different settings from the BorderStyle property in other controls and in forms. The BorderStyle property of a shape or line control serves a different purpose and uses different settings from the BorderStyle property on other objects. For shape and line controls, the BorderStyle property works like the DrawStyle property as described in this section. For forms and other controls, the BorderStyle property determines whether the control or form has a border and if so, whether the border is fixed or sizable.

116

Solid and Inside Solid Styles

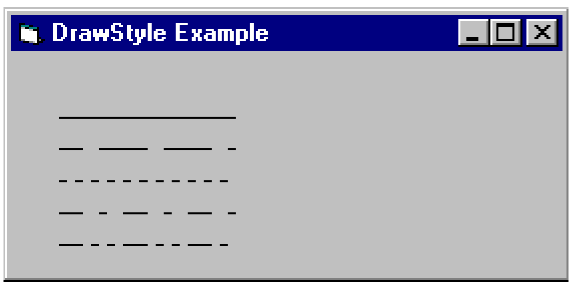
The inside solid style (DrawStyle or BorderStyle = 6) is nearly identical to the solid style. They both create a solid line. The difference between these settings becomes apparent when you use a wide line to draw a box or a shape control. In these cases, the solid style draws the line half inside and half outside the box or shape. The inside solid style draws the line entirely inside the box or shape. See “Drawing Boxes,” earlier in this chapter, to see how to draw a box.

The following procedure demonstrates all of the supported settings of the DrawStyle property by creating a loop in which the setting goes from 0 to 6, one step at a time. The results are shown in Figure 12.20.

```
Private Sub Form_Click ()
    Dim I As Integer, Y As Long
    For I = 0 To 6
        DrawStyle = I
        Y = (200 * I) + 1000
        Line (200, Y)–(2400, Y)
    Next I
End Sub
```

117

Figure 12.20   The effects of changing the DrawStyle property



49

Controlling Display Using DrawMode

The DrawMode property determines what happens when you draw one pattern on top of another. Although changing the DrawMode property usually has some effect (especially with color systems), it is often not necessary to use this property when you are drawing on a blank or pure white background, or on a background of undifferentiated color.

You can set DrawMode to a value from 1 to 16. Common settings appear in the following table.

Setting	Description
4	Not Copy Pen. Draws the inverse of the line pattern, regardless of what is already there.
7	Xor Pen. Displays the difference between the line pattern and the existing display, as explained later in this section. Drawing an object twice with this mode restores the background precisely as it was.
11	No operation. In effect, this turns drawing off.
13	Copy Pen (default). Applies the line’s pattern, regardless of what is already there.

118

The Xor Pen

A DrawMode setting of 7 is useful for animation. Drawing a line twice restores the existing display precisely as it was before the line was drawn. This makes it possible to create one object that “moves over” a background without corrupting it, because you can restore the background as you go. Most modes are not guaranteed to preserve the old background.

For example, the following code moves a circle every time the mouse is clicked. No matter what pattern was underneath the circle, it gets restored.

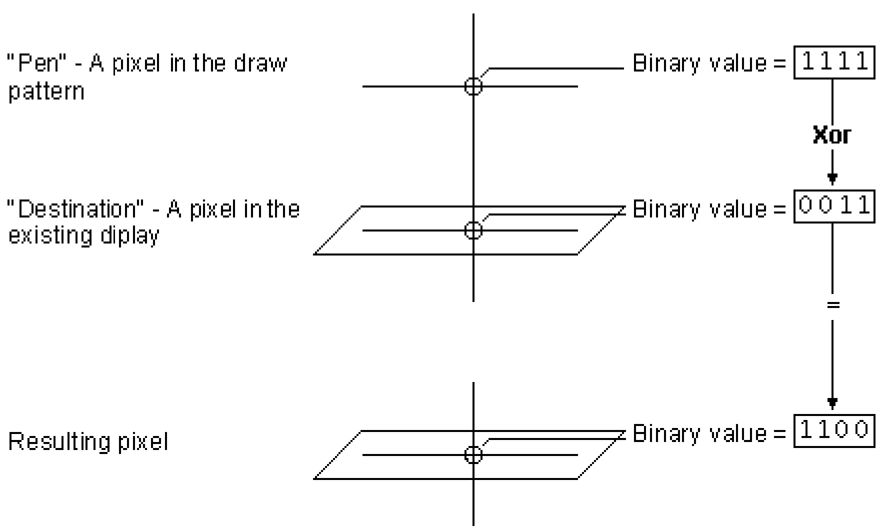
```
Private Sub Form_Click ()
    ForeColor = 255 : DrawMode = 7
    Circle (CurrentX, CurrentY), 1000
    CurrentX = CurrentX + 220
    CurrentY = CurrentY + 220
    Circle (CurrentX, CurrentY), 1000
End Sub
```

119

The Xor Pen draw mode (and most of the other DrawMode settings) works by comparing each individual pixel in the draw pattern (called the “Pen”) and the corresponding pixel in the existing area (called the “Destination”). On monochrome systems, the pixel is turned either on or off, and Visual Basic performs a simple logical comparison: It turns a pixel on if either the Pen or Destination pixel is on, but not if both are on.

In color systems, each pixel is assigned a color value. For DrawMode settings such as Xor Pen, Visual Basic compares each corresponding pair of pixels in the Pen and Destination and performs a binary (bitwise) comparison. The result determines the color value of the resulting pixel, as shown in Figure 12.21.

Figure 12.21 Using the Xor Pen to set the binary value of a pixel in a line



Creating Graphics When a Form Loads

When creating graphics that appear on a form when it loads, consider placing the graphics methods in the Form_Paint event. Form_Paint graphics will get repainted automatically in every paint event. If you place graphics in the Form_Load event, set the AutoRedraw property on the form to True. In this case, Form_Load should show the form, then draw the graphics. Remember, forms are not visible during the Form_Load event. Because Visual Basic does not process graphics methods on a form that is not visible, graphics methods in the Form_Load event are ignored unless AutoRedraw is set to True.

Working with Color

Visual Basic uses a consistent system for all color properties and graphics methods. A color is represented by a Long integer, and this value has the same meaning in all contexts that specify a color.

Specifying Colors at Run Time

There are four ways to specify a color value at run time:

- Use the RGB function.
- Use the QBColor function to choose one of 16 Microsoft QuickBasic colors.
- Use one of the intrinsic constants listed in the Object Browser.
- Enter a color value directly.

51

This section discusses how to use the RGB and QBColor functions as simple ways to specify color. See “Using Color Properties" later in this chapter for information on using constants to define color or directly entering color values.

Using the RGB Function

You can use the RGB function to specify any color.

▯ To use the RGB function to specify a color

1 Assign each of the three primary colors (red, green, and blue) a number from 0 to 255, with 0 denoting the least intensity and 255 the greatest.

2 Give these three numbers as input to the RGB function, using the order red-green-blue.

3 Assign the result to the color property or color argument.

52

Every visible color can be produced by combining one or more of the three primary colors. For example:

```
' Set background to green.
Form1.BackColor = RGB(0, 128, 0)
```

```
' Set background to yellow.
Form2.BackColor = RGB(255, 255, 0)
' Set point to dark blue.
PSet (100, 100), RGB(0, 0, 64)
```

120

Using Color Properties

Many of the controls in Visual Basic have properties that determine the colors used to display the control. Keep in mind that some of these properties also apply to controls that aren't graphical. The following table describes the color properties.

Property	Description
BackColor	Sets the background color of the form or control used for drawing. If you change the BackColor property after using graphics methods to draw, the graphics are erased by the new background color.
ForeColor	Sets the color used by graphics methods to create text or graphics in a form or control. Changing ForeColor does not affect text or graphics already created.
BorderColor	Sets the color of the border of a shape control.
FillColor	Sets the color that fills circles created with the Circle method and boxes created with the Line method.

121

Defining Colors

The color properties can use any of several methods to define the color value. The RGB function described in “Working with Color” is one way to define colors. This section discusses two more ways to define colors:

- Using defined constants
- Using direct color settings

53

Using Defined Constants

You don’t need to understand how color values are generated if you use the intrinsic constants listed in the Object Browser. In addition, intrinsic constants do not need to be declared. For example, you can use the constant vbRed whenever you want to specify red as a color argument or color property setting:

```
BackColor = vbRed
```

122

Using Direct Color Settings

Using the RGB function or the intrinsic constants to define color are indirect methods. They are indirect because Visual Basic interprets them into the single approach it uses to represent color. If you understand how colors are represented in Visual Basic, you can assign numbers to color properties and arguments that specify color directly. In most cases, it’s much easier to enter these numbers in hexadecimal.

The valid range for a normal RGB color is 0 to 16,777,215 (&HFFFFFF&). Each color setting (property or argument) is a 4-byte integer. The high byte of a number in this range equals 0. The lower 3 bytes, from least to most significant byte, determine the amount of red, green, and blue, respectively. The red, green, and blue components are each represented by a number between 0 and 255 (&HFF).

Consequently, you can specify a color as a hexadecimal number using this syntax:

&*HBBGGRR***&** 123

The *BB* specifies the amount of blue, *GG* the amount of green, and *RR* the amount of red. Each of these fragments is a two-digit hexadecimal number from 00 to FF. The median value is 80. Thus, the following number specifies gray, which has the median amount of all three colors:

```
&H808080&
```
124

Setting the most significant bit to 1 changes the meaning of the color value: It no longer represents an RGB color, but an environment-wide color specified through the Windows Control Panel. The values that correspond to these system-wide colors range from &H80000000 to &H80000015.

Note Although you can specify over 16 million different colors, not all systems are capable of displaying them accurately. For more information on how Windows represents colors, see “Working with 256 Colors” later in this chapter.

125

Using System Colors

When setting the colors of controls or forms in your application, you can use colors specified by the operating system instead of specific color values. If you specify system colors, when users of your application change the values of system colors on their computers, your application automatically reflects the user-specified color values.

Each system color has both a defined constant and a direct color setting. The high byte of direct color settings for system colors differs from those of normal RGB colors. For RGB colors, the high byte equals 0 whereas for system colors the high byte equals 8. The rest of the number refers to a particular system color. For example, the hexadecimal number used to represent the color of an active window caption is &H80000002&.

When you select color properties at design time with the Properties window, selecting the System tab lets you choose system settings, which are automatically converted into the hexadecimal value. You can also find the defined constants for system colors in the Object Browser.

Working with 256 Colors

Visual Basic supports 256 colors on systems with video adapters and display drivers that handle 256 or more colors. The ability to display 256 simultaneous colors is particularly valuable in multimedia applications or applications that need to display near –photographic-quality images.

You can display 256-color images and define up to 256 colors for graphics methods in:

- Forms
- Picture boxes
- Image controls (display images only)

54

Note Support for 256 colors does not apply to Windows metafiles. Visual Basic displays metafiles using the default palette of 16 VGA colors.

126

Color Palettes

Color palettes provide the basis for 256-color support in Visual Basic applications. In discussing palettes, it's important to understand the relationship between different palette types. The *hardware palette* contains 256 entries defining the actual RGB values that will be displayed on screen. The *system halftone palette* is a predefined set of 256 RGB values made available by Windows itself. A *logical palette* is a set of up to 256 RGB values contained within a bitmap or other image.

Windows can draw using the 256 colors in the hardware palette. Twenty of these 256 colors, called *static colors*, are reserved by the system and cannot be changed by an application. Static colors include the 16 colors in the default VGA palette (the same as the colors defined by Visual Basic's QBColor function), plus four additional shades of gray. The system halftone palette always contains these static colors.

The foreground window (the window with focus) determines the 236 nonstatic colors in the hardware palette. Each time the hardware palette is changed, all background windows are redrawn using these colors. If the colors in a background window's logical palette don't perfectly match those currently in the hardware palette, Windows will assign the closest match.

Displaying 256-Color Images

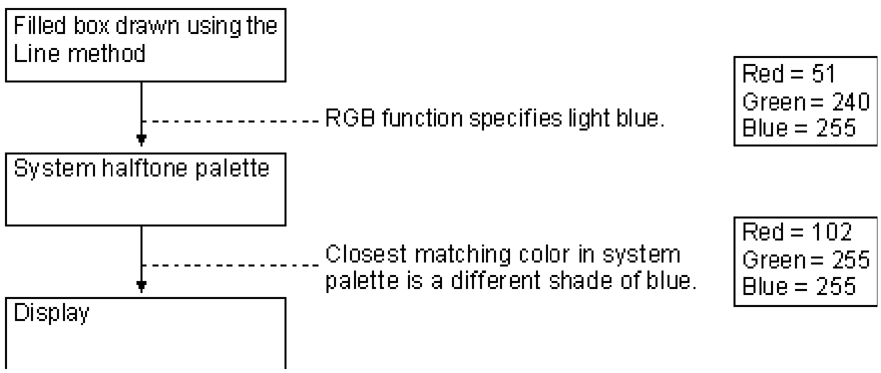
Forms, picture boxes, and image controls automatically display images in 256 colors if the user's display hardware and software can support that many colors on screen. If the user's system supports fewer colors than the image, then Visual Basic will map all colors to the closest available.

On true-color (16-million color) displays, Visual Basic always uses the correct color. On monochrome or 16-color displays, Visual Basic will dither background colors and colors set with the FillColor property. *Dithering* is a process used to simulate colors not available from the video adapter and display driver.

Drawing with Color Palettes

With 256-color video drivers, you can use up to 256 colors with graphics methods. By default, the 256 colors available in Visual Basic are those in the system halftone palette. Although you can specify an exact color using the RGB function, the actual color displayed will be the closest match from the halftone palette, as shown in Figure 12.22.

Figure 12.22   Color matching from a specified color to the display



55

Although the default palette for Visual Basic is the system halftone palette, you can also control the display of colors with the PaletteMode and Palette properties of forms, user controls, and user documents. In this case, the color match is much the same, except that colors will be matched to the closest color in the hardware palette.

For More Information   To learn more about the Palette and PaletteMode properties, see "Managing Multiple Color Palettes" later in this chapter.

127

Managing Multiple Color Palettes

When you work with color palettes, keep in mind that many displays can display only 256 colors simultaneously on the screen.

This limitation becomes important when you use more than one color palette in your application. For example, on a single form, you might display a 256-color bitmap in an image control while displaying a second image in a picture box. If the logical palettes of these two images don't contain exactly the same 256 colors, Windows must decide which logical palette places its colors in the hardware palette first. Remember: The hardware palette determines what actually appears on the screen.

A similar situation occurs when your Visual Basic application has two or more forms with differing logical palettes. As each form receives focus, its logical palette controls the hardware palette. This can often result in a less than optimal display on 256-color systems. As a Visual Basic programmer, you can control the hardware palette by using the PaletteMode property.

The PaletteMode Property

When designing applications that may run on 256-color systems, you can control the way that Windows chooses the display colors by setting the PaletteMode property of a form, user control, or user document. (User controls and user documents are only available in the Professional and Enterprise editions.) All controls contained on the form, user control, or user document will be displayed based on the PaletteMode. The following table shows the available PaletteMode settings:

Mode	Constant	Applies to
Halftone	vbPaletteModeHalftone	Forms, User Controls, User Documents
UseZOrder	vbPaletteModeUseZOrder	Forms, User Controls, User Documents
Custom	vbPaletteModeCustom	Forms, User Controls, User Documents
Container	vbPaletteModeContainer	User Controls
None	vbPaletteModeNone	User Controls

128

The PaletteMode property only applies to 256-color displays. On high-color or true-color displays, color selection is handled by the video driver using a palette of 32,000 or 16 million colors respectively. Even if you're programming on a system with a high-color or true-color display, you still may want to set the PaletteMode, because many of your users may be using 256-color displays.

The PaletteMode property can be set at design time through the Properties window, or changed at run time via code. The Palettes sample application demonstrates the effects of displaying images with different palettes using several different PaletteMode settings.

Halftone PaletteMode

The default mode for forms and user documents is Halftone. In this mode, any controls, images contained on the form, or graphics methods draw using the system halftone palette.

Halftone mode is a good choice in most cases because it provides a compromise between the images in your form, and colors used in other forms or images. It may, however, result in a degradation of quality for some images. For example, an image with a palette containing 256 shades of gray may lose detail or display unexpected traces of other colors.

UseZOrder PaletteMode

Z-order is a relative ordering that determines how controls overlap each other on a form. When the PaletteMode of the form with the focus is set to UseZOrder, the palette of the topmost control always has precedence. This means that each time a new control becomes topmost (for instance, when you load a new image into a picture box), the hardware palette will be remapped. This will often cause a side effect known as palette flash: The display appears to flash as the new colors are displayed, both in the current form and in any other visible forms or applications.

Although the UseZOrder setting provides the most accurate color rendition, it comes at the expense of speed. Additionally, this method can cause the background color of the form or of controls that have no image to appear dithered. Setting the PaletteMode to UseZOrder is the best choice when accurate display of the topmost image outweighs the annoyance of palette flash, or when you need to maintain backward compatibility with earlier versions of Visual Basic.

Custom PaletteMode

If you need more precise control over the actual display of colors, you can use a 256-color bitmap to define a custom palette. To do this, assign a 256-color bitmap (.bmp) to the Palette property of the form and set the PaletteMode property to Custom. The bitmap doesn't have to be very large; even a single pixel can define up to 256 colors for the form or picture box. This is because the logical palette of a bitmap can list up to 256 colors, regardless of whether all those colors appear in the bitmap.

Visual Basic ships three bitmaps with color palettes you can load into forms and picture boxes, or you can use any 256-color bitmap. The following table describes these bitmaps.

Device-independent bitmap (.dib) file	Palette description
Rainbow.dib	Standard range of all hues.
Pastel.dib	Lighter hues, primarily blues.
Bright.dib	Bright shades of all hues.

129

As with the default method, colors that you define using the RGB function must also exist in the bitmap. If the color doesn't match, it will be mapped to the closest match in the logical palette of the bitmap assigned to the Palette property.

To set the Custom PaletteMode at run time, add the following code to the Form_Load event (assuming that the image containing your chosen palette has been assigned to a Image control named Image1):

```
' Assign the palette from Image1 to the form.
Form1.Palette = Image1.Picture
' Use the Custom mode.
Form1.PaletteMode = vbPaletteModeCustom
```

130

Alternatively, you can use the Picture object to achieve the same effect without the extra Image control:

```
Dim objPic As Picture
Set objPic = LoadPicture(App.Path & "\Pastel.dib")
' Assign picture object's palette to the form.
Form1.Palette = objPic
' Use the Custom mode.
Form1.PaletteMode = vbPaletteModeCustom
```

131

The Custom PaletteMode is your best choice when you want to maintain a uniform palette throughout your application.

Note Using the Custom PaletteMode can also improve the performance of your application in cases where you aren't using any 256-color graphics. If you set the PaletteMode of a form to Custom and leave the Palette property blank, your form will load faster because no palette matching will occur. 132

For More Information To learn more about the Picture object, see "Using the Picture Object" later in this chapter. 133

Other Palette Modes

Two additional PaletteMode settings are available when creating user controls: Container and None. The Container mode maps the palette of the user control and any contained controls to the ambient palette of the container (form or user document) at run time. If the container doesn't supply an ambient palette, the Halftone mode will be invoked. Because you may not know in advance where your user control may be deployed, this mode can prevent your control from conflicting with other palette handling methods.

The None mode does just what you might expect: It eliminates palette handling altogether. When creating a user control that doesn't display images or graphics, setting PaletteMode to None improves performance by eliminating the added overhead of handling palette messages.

Using the Picture Object

The Picture object is similar in some respects to the Printer object — you can't see it, but it's useful nonetheless. You could think of the Picture object as a invisible picture box that you can use as a staging area for images. For example, the following code loads a Picture object with a bitmap and uses that bitmap to set the Picture property of a picture box control:

```
Private Sub Command1_Click()
    Dim objPic As Picture
    Set objPic = LoadPicture("Butterfly.bmp")
    Set Picture1.Picture = objPic
End Sub
```

134

The Picture object supports bitmaps, GIF images, JPEG images, metafiles, and icons.

Using Arrays of Picture Objects

You can also use an array of Picture objects to keep a series of graphics in memory without using a form that contains multiple picture box or image controls. This is convenient for creating animation sequences or other applications where rapid image changes are required. Declare the array at the module level:

```
Dim objPics(1) As Picture
```

Add the following code to the Form_Load event: 135

```
' Load bitmaps int the Picture object array.
Set objPics(0) = LoadPicture("Butterfly1.bmp")
Set objPics(1) = LoadPicture("Butterfly2.bmp")
```

Then in Timer event you can cycle the images: 136

```
Static intCount As Integer
If intCount = 0 Then
    intCount = 1
Else
    intCount = 0
End If
' Use the PaintPicture method to display the bitmaps
'    on the form.
PaintPicture objPics(intCount), 0, 0
```

By adding a loop to increment the x and y coordinates, you could easily make the 137
butterfly bitmaps “fly” across the form.

Using the Picture Object Instead of the Windows API

There are lots of things you can do with bitmaps, icons, or metafiles in the Windows API, but the Picture object already does most of them for you. This means that you are better off using the Picture object instead of the Windows API whenever possible. The Picture object also allows you to use .jpeg and .gif files, whereas the Windows API does not.

There is no direct relationship between a Picture.Handle and a PictureBox.hDC. The hDC property of the picture box is the handle provided by the operating system to the device context of the picture box control. The Handle property of the Picture object is actually the handle of the GDI object that is contained in the Picture object.

There are now two completely different ways to paint graphics on a window (or blit). You can use BitBlt or StretchBlt on the hDC of an object, or you can use the PaintPicture method on the Picture object or property. If you have an Image control, you can only use PaintPicture because Image controls do not have an hDC.

For More Information For more information about the Windows API, see "Accessing DLLs and the Windows API."

138

Printing

Printing is one of the most complex tasks a Windows – based application performs. Good results depend on all parts of the process working together. Poor results can arise from problems in your application, variations in printer drivers, or limited printer capabilities. Although it is a good idea to test your application with commonly used printers and printer drivers, you can’t test all the possible combinations users may have.

Printing from your application involves these three components:

- The code in your application that starts the printing process.
- The printer drivers installed on both your system and the systems of users of your application.
- The capabilities of the printers available to users of your application.

56

The code in your application determines the type and quality of print output available from your application. But the users' printer drivers and printers also impact print quality. This section deals with enabling printing from a Visual Basic application. For information on printing from the Visual Basic development environment, see "Printing Information in the Immediate Window" in "Debugging Your Code and Handling Errors."

Printing from an Application

Visual Basic provides three techniques for printing text and graphics.

- You can produce the output you want on a form and then print the form using the PrintForm method.
- You can send text and graphics to a printer by setting the default printer to a member of the Printers collection.
- You can send text and graphics to the Printer object and then print them using the NewPage and EndDoc methods.

57

This section examines the advantages and disadvantages of these three approaches.

Using the PrintForm Method

The PrintForm method sends an image of the specified form to the printer. To print information from your application with PrintForm, you must first display that information on a form and then print that form with the PrintForm method. The syntax is as follows:

[*form*.]**PrintForm**

139

If you omit the form name, Visual Basic prints the current form. PrintForm prints the entire form, even if part of the form is not visible on the screen. If a form contains graphics, however, the graphics print only if the form's AutoRedraw property is set to True. When printing is complete, PrintForm calls the EndDoc method to clear the printer.

For example, you could send text to a printer by printing it on a form and then calling PrintForm with the following statements:

```
Print "Here is some text."
PrintForm
```

140

The PrintForm method is by far the easiest way to print from your application. Because it may send information to the printer at the resolution of the user's screen (typically 96 dots per inch), results can be disappointing on printers with much higher resolutions (typically 300 dots per inch for laser printers). The results may vary depending on objects on your form.

Using the Printers Collection

The Printers collection is an object that contains all the printers that are available on the operating system. The list of Printers are the same as those available in the Print Setup dialog box or the Windows Control Panel. Each printer in the collection has a unique index for identification. Starting with 0, each printer in the collection can be referenced by its number.

Regardless of which printing method you use, all printed output from a Visual Basic application is directed to the Printer object, which initially represents the default printer specified in the Windows Control Panel. However, you can set the default printer to any one member in the Printers collection.

To select the printer from the collection, use the following syntax:

Set Printer = Printers(*n*)

141

The following statements print the device names of all the printers on the operating system to the Debug window:

```
Private Sub Command1_Click()
Dim x As Printer
    For Each x In Printers
        Debug.Print x.DeviceName
    Next
End Sub
```

142

Note You cannot create new instances of the Printer object in code, and you cannot directly add or remove printers from the Printers collection. To add or remove printers on your system, use the Windows Control Panel.

143

Using the Printer Object

The Printer object is a device-independent drawing space that supports the Print, PSet, Line, PaintPicture, and Circle methods to create text and graphics. You use these methods on the Printer object just as you would on a form or picture box. The Printer object also has all the font properties described earlier in this chapter. When you finish placing the information on the Printer object, you use the EndDoc method to send the output to the printer. When applications close, they automatically use the EndDoc method to send any pending information on the Printer object.

The Printer object provides the best print quality across a variety of printers because Windows translates text and graphics from the device-independent drawing space of the Printer object to best match the resolution and abilities of the printer. You can also print multiple-page documents by using the NewPage method on the Printer object.

The main drawback to using the Printer object is the amount of code required to get the best results. Printing bitmaps on the Printer object also takes time and can therefore slow the performance of the application.

Printing with the Printer Object

There are several ways to place text and graphics on the Printer object. To print with the Printer object, do any of the following:

- Assign the specific member of the Printers collection to the Printer object if you want to print to a printer other than the default printer.
- Put text and graphics on the Printer object.
- Print the contents of the Printer object with the NewPage or EndDoc method. 58

Printer Object Properties

The properties of the Printer object initially match those of the default printer set in the Windows Control Panel. At run time, you can set any of the Printer object properties, which include: PaperSize, Height, Width, Orientation, ColorMode, Duplex, TrackDefault, Zoom, DriverName, DeviceName, Port, Copies, PaperBin, and PrintQuality. For more details and syntax for these methods, see the *Language Reference* in Books Online.

If the TrackDefault property is True and you change the default printer in the Windows Control Panel, the Printer object property values will reflect the properties of the new default printer.

You cannot change some properties in the middle of a page once a property has been set. Changes to these properties will only affect subsequent pages. The following statements show how you can print each page using a different print quality:

```
For pageno = 1 To 4
    Printer.PrintQuality = -1 * pageno
    Printer.Print "The quality of this page is"; pageno
    Printer.NewPage
Next
```
144

Print quality values can range from – 4 to – 1, or a positive integer corresponding to the print resolution in dots per inch (DPI). For example, the following code would set the printer’s resolution to 300 DPI:

```
Printer.PrintQuality = 300
```
145

Note The effect of Printer property values depends on the driver supplied by the printer manufacturer. Some property settings may have no effect, or several different property settings may all have the same effect. Settings outside the accepted range may or may not produce an error. For more information on specific drivers, see the manufacturer's documentation.

146

Scale Properties

The Printer object has these scale properties:

- ScaleMode
- ScaleLeft and ScaleTop
- ScaleWidth and ScaleHeight
- Zoom

59

The ScaleLeft and ScaleTop properties define the x- and y-coordinates, respectively, of the upper-left corner of a printable page. By changing the values of ScaleLeft and ScaleTop, you can create left and top margins on the printed page. For example, you can use ScaleLeft and ScaleTop to center a printed form (PFrm) on the page using these statements:

```
Printer.ScaleLeft = -((Printer.Width - PFrm.Width) / 2)
Printer.ScaleTop = -((Printer.Height - PFrm.Height) _
    / 2)
```

147

Many printers support the Zoom property. This property defines the percentage by which output is scaled. The default value of the Zoom property is 100, indicating that output will be printed at 100 percent of its size (actual size). You can use the Zoom property to make the page you print smaller or larger than the actual paper page. For example, setting Zoom to 50 makes your printed page appear half as wide and half as long as the paper page. The following syntax sets the Zoom property to half the size of the default Printer object:

```
Printer.Zoom = 50
```

148

Positioning Text and Graphics

You can set CurrentX and CurrentY properties for the Printer object, just as you can for forms and picture boxes. With the Printer object, these properties determine where to position output on the current page. The following statements set drawing coordinates to the upper-left corner of the current page:

```
Printer.CurrentX = 0
Printer.CurrentY = 0
```

149

You can also use the TextHeight and TextWidth methods to position text on the Printer object. For more information on using these text methods, see "Displaying Print Output at a Specific Location" earlier in this chapter.

Printing Forms on the Printer Object

You may want your application to print one or more forms along with information on those forms, especially if the design of the form corresponds to a printed document like an invoice or a time card. For the easiest way to do this, use the PrintForm method. For the best quality on a laser printer use the Print and graphics methods with the Printer object. Keep in mind that using the Printer object takes more planning, because you must recreate the form on the Printer object before you print.

Recreating a form on the Printer object may also require recreating:

- The outline of the form, including title and menu bars.
- The controls and their contents, including text and graphics.
- The output of graphics methods applied directly to the form, including the Print method.

60

The extent to which you recreate these elements on the Printer object depends on your application and how much of the form you need to print.

Recreating Text and Graphics on a Form

When creating text and graphics on a form using the Print, Line, Circle, PaintPicture, or PSet methods, you may also want a copy of this output to appear on the Printer object. The easiest way to accomplish this is to write a device-independent procedure to recreate the text and graphics.

For example, the following procedure uses the PaintPicture method to print a form or control's Picture property to any output object, such as a printer or another form:

```
Sub PrintAnywhere (Src As Object, Dest As Object)
    Dest.PaintPicture Src.Picture, Dest.Width / 2, _
        Dest.Height / 2
    If Dest Is Printer Then
        Printer.EndDoc
    End If
End Sub
```

150

You then call this procedure and pass it the source and destination objects:

```
PrintAnywhere MyForm, Printer
PrintAnywhere MyForm, YourForm
```

151

Printing Controls on a Form

The Printer object can receive the output of the Print method and the graphics methods (such as the Line or PSet method). But you cannot place controls directly on the Printer object. If your application needs to print controls, you must either write procedures that redraw each type of control you use on the Printer object, or use the PrintForm method.

Printing the Contents of the Printer Object

Once you have placed text and graphics on the Printer object, use the EndDoc method to print the contents. The EndDoc method advances the page and sends all pending output to the spooler. A *spooler* intercepts a print job on its way to the printer and sends it to disk or memory, where the print job is held until the printer is ready for it. For example:

```
Printer.Print "This is the first line of text in _
     a pair."
Printer.Print "This is the second line of text in _
     a pair."
Printer.EndDoc
```

152

Note Visual Basic automatically calls EndDoc if your application ends without explicitly calling it.

153

Creating Multiple-Page Documents

When printing longer documents, you can specify in code where you want a new page to begin by using the NewPage method. For example:

```
Printer.Print "This is page 1."
Printer.NewPage
Printer.Print "This is page 2."
Printer.EndDoc
```

154

Canceling a Print Job

You can terminate the current print job by using the KillDoc method. For example, you can query the user with a dialog box to determine whether to print or terminate a document:

```
Sub PrintOrNot()
     Printer.Print "This is the first line to _
          illustrate KillDoc method"
     Printer.Print "This is the second line to _
          illustrate KillDoc method"
     Printer.Print "This is the third line to _
          illustrate KillDoc method"
     If vbNo = MsgBox("Print this fine document?", _
               vbYesNo) Then
          Printer.KillDoc
     Else
          Printer.EndDoc
     End If
End Sub
```

155

If the operating system's Print Manager is handling the print job, the KillDoc method deletes the entire job you sent to the printer. However, if the Print Manager is not controlling the print job, page one may have already been sent to the printer, and will be unaffected by KillDoc. The amount of data sent to the printer varies slightly among printer drivers.

Note You cannot use the KillDoc method to terminate a print job that was initiated with the PrintForm method.

156

Trapping Printer Errors

Trappable run-time errors may occur while printing. The following table lists some examples that may be reported:

Error number	Error message
396	**Property cannot be set within a page.** This error occurs when the same property is set differently on the same page.
482	**Printer Error.** Visual Basic reports the error whenever the printer driver returns an error code.
483	**Printer driver does not support the property.** This error occurs when attempting to use a property that is not supported by the current printer driver.
484	**Printer driver unavailable.** This error occurs when the WIN.INI printer information is missing or insufficient.

157

Note Printer errors do not always occur immediately. If a statement causes a printer error, the error may not be raised until execution of the next statement that addresses that printer.

158

operations in you

For More Information For a detailed discussion on run-time errors, see "Debugging Your Code and Handling Errors."

159