

A P P E N D I X

C

Using Multimedia Elements to Enhance Applications

- Images
- Wipes
- Waveform audio
- MIDI
- MCI commands
- Video for Windows

When you check out an application, especially a game, what often catches your eye is multimedia. The images, sounds, and animation are “eye candy” that enhance the application’s user interface and make the application more approachable and fun to the average user. You can achieve highly desirable effects by combining some or all the following elements:

- Cool graphics
- Great sounds
- Dynamic animation
- Stunning digital video

This appendix explores the basic tools and the techniques you need to incorporate these elements into your applications. Although these exciting elements can add a lot to your application, they are not substitutes for quality content and functionality. But even a not-so-great application can look attractive when enhanced with multimedia elements.

Enhancing Applications with Images

Most of the information you need in order to add images to your application is covered in Chapter 6, *Drawing with Visual Basic*, and Chapter 7, *Manipulating Pixels and Color with Visual Basic*. In this appendix, you’ll learn another useful technique, namely, how to use the `PaintPicture` method to create smooth transitions between images.

Images come in many flavors and formats. Depending on what you plan to do with them, you might need to do some conversion before you can actually use them. Also, you should store images based on what you intend to do with them. If you want your images to be accessed on the Internet, you should probably consider reducing their size by compressing them. If the users of your application will have True Color displays, you can use high-resolution, True Color images.

Image Quality

The resolution of your screen can profoundly affect the image that you display. Each individual pixel on the screen corresponds to a value in the computer’s video memory. In an image, the number of pixels determines the *spatial resolution* (or simply, resolution), and the number of colors determines the *color resolution* (or color

depth). The same two resolutions apply to the display as well. The *display resolution* is determined by number of pixels per row and number of rows that can be displayed on a monitor, and its color resolution is the number of colors it can display.

The image and display resolutions are independent; the only connection is that if the image is too large for a given monitor resolution, the image won't be displayed in its entirety. In general, size your images so that they will fit nicely on a monitor with resolution of 800×600 . The lowest common denominator is 640×480 , but this screen size is gradually being abandoned. Most systems sold today, even notebooks, have a screen resolution of 800×600 .

Since color resolution refers to the number of colors in an image, it's not related to screen resolution or the size of an image. The more colors an image contains, the smoother it looks. As with the physical resolution of an image, the more colors it contains, the more disk space it requires for storage and the more time it takes to load. Screen resolution and color resolution are independent of each other, but usually one helps the other. Games are usually displayed on low-resolution monitors (or a small area of the screen) because updating too many pixels is an enormous computational burden (and games are not normally played on top-of-the-line computers). To make up for the low resolution, game designers use many colors.

True Color and Palettes

A True Color image yields the highest quality, but what if the user's computer can't display all the colors in such an image? As you probably know, many systems can display only 256 colors. To address a larger number of users, your applications should not fall apart when executed on these systems. Multimedia systems, however, support True Color, and it's not unreasonable to request that your multimedia-enhanced applications run on systems that support more than 256 colors.

Systems that can't display more than 256 colors use a palette of 256 colors. Only colors present in this palette can be displayed. Since the 256 colors aren't fixed, there's an optimal palette for each image. This palette contains the 256 colors that best describe that image. If you display another image with similar colors, Visual Basic won't have a problem displaying it because most of the colors it contains are already on the palette or can be approximated. However, if you attempt to display another image with drastically different colors, Visual Basic has three choices:

1. Create a common palette with the colors of both images.
2. Switch to the appropriate palette, depending on which image has the focus.
3. Use halftone patterns to approximate the colors of both images.

NOTE

A *halftone* consists of tiny, evenly spaced spots of variable diameter that, when printed (on screen or on paper), visually blur together to appear as shades of gray. On the monitor, you can't have dots of variable diameter, so it's the number of dots of each color placed next to each other that produce an approximate color.

The display method used depends on the setting of the Form's `PaletteMode` property, which can take one of the values shown in Table C.1.

TABLE C.1: Values of the `PaletteMode` Property

VALUE	TYPE	DESCRIPTION
0	Halftone	Halftone patterns are used to accommodate the colors of both images
1	UseZOrder	The palette of the image in front of the ZOrder is used to render the colors of both images
2	Custom	A user-specified palette is used to render the colors of both images

PaletteMode 0 When `PaletteMode` is 0, Visual Basic uses halftone patterns to display the colors of all images, as best as it can, every time a new image is loaded. All images are displayed with approximate colors, and none are displayed as they would appear on their own.

PaletteMode 1 When `PaletteMode` is 1, Visual Basic uses the palette of the image in front of the ZOrder for all the images on the Form. This setting allows you to control from within your application which image will be displayed with its colors, without regard for the other images (whose colors may change drastically).

PaletteMode 2 When `PaletteMode` is 2, you can assign the palette that will be used to render the colors of all images on the Form. This is equivalent to changing the ZOrder of the various images, with one exception: the common palette can belong to an image that isn't even displayed. You can use a generic palette with many colors (such as the palettes that come with Visual Basic) for all the images. Again, the image whose colors are closest to the palette's colors will be displayed best.

NOTE

PaletteMode is a Form property and it determines how the images on the Form or the Form's controls will be displayed. `PictureBox` and `ImageBox` controls don't have a `PaletteMode` or `Palette` property.

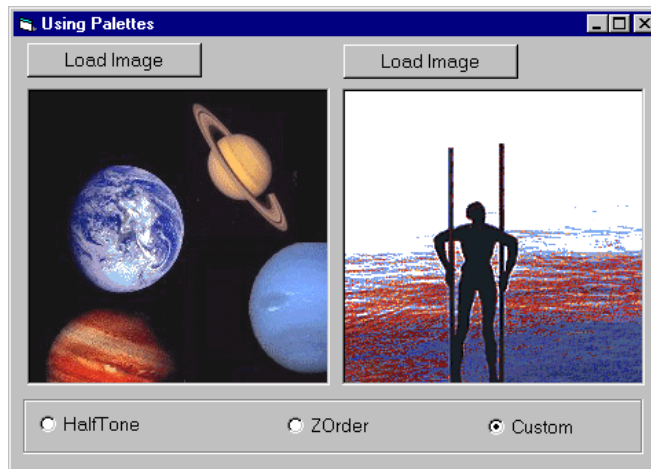
VB6 at Work: The Palettes Project

To demonstrate the `PaletteMode` and `Palette` properties, I have included the Palettes application in the projects folder for this appendix, shown in Figure C.1. The Load Image buttons above the PictureBox controls let you load a new image to the corresponding PictureBox. You should try both True Color and palette images.

The three option buttons at the bottom of the Form set the `PaletteMode` property of the Form. After setting the mode, you can click one of the two images. If the `PaletteMode` is set to 1 (`ZOrder`), the image you click is brought to the front of the `ZOrder`. If the `PaletteMode` is 2 (`Custom`), the palette of the image you click is assigned to the Form's `Palette` property so that both images are rendered with this palette. Experiment with the various settings and various types of images to understand how Visual Basic handles multiple palettes and True Color images on palette systems.

FIGURE C.1:

The Palettes application lets you experiment with the settings of the `PaletteMode` property.



The three `OptionButton` controls form an array. The code behind the option buttons changes the `PaletteMode` property of the Form and is shown next.

Code C.1: The Click Event Handler Code for the `OptionButton` Controls

```
Private Sub Palette_Click(Index As Integer)

    Form1.PaletteMode = Index
    Form1.Refresh

End Sub
```

The other subroutine worth mentioning is the PictureBox control's Click event handler.

Code C.2: The Click Event Handler Code for the PictureBox Control

```
Private Sub Picture1_Click()

    If Form1.PaletteMode = 1 Then
        Picture1.ZOrder 0
        Picture2.ZOrder 1
    End If
    If Form1.PaletteMode = 2 Then
        Form1.Palette = Picture1.Image
    End If

End Sub
```

The action of this subroutine depends on the setting of the PaletteMode property. To test the Palettes application, set your computer's display to 256 colors. On a True Color system, all images are displayed in their original colors. If the PaletteMode is 1 (ZOrder), the code moves the PictureBox control to the front of the ZOrder. If PaletteMode is 2 (Custom), the palette of the control that's clicked becomes the Form's palette and both images are rendered with this palette. You can load the palettes that come with Visual Basic (the files Bright.dib, Rainbow.dib, and Pastel.dib) in one of the PictureBox controls and then apply this palette to another image. You can also add a third PictureBox control or assign an image to the Form to see how Visual Basic handles multiple images with different palette requirements.

Image Formats

A graphics file is the format in which graphics data are stored. Because programs try to find an optimal method for storing their data, a graphics file format can be complex. Three factors are taken into account:

1. The speed at which the image data can be accessed
2. The total disk space required
3. The applications that can access the image data

An important aspect of all image file formats is compression. Images easily can be compressed because they contain a lot of redundant information. For instance, a long stripe of blue pixels doesn't need to be stored as a sequence of identical

numbers. The most common compression algorithms store the value of one pixel and how many times this value is repeated. This type of compression is called *Run Length Encoding (RLE)*. RLE's basic benefit is speed and its compressed images can be decompressed easily and quickly. However, it isn't the most efficient compression scheme.

Other file formats deploy more complicated and efficient compression schemes. Keep in mind, the more complicated the algorithms used for image compression, the more calculations they require, which means additional loading time.

The most common formats used in today's applications include bitmap, graphics interchange, icon, cursor, and metafile.

Windows Bitmap

The *Microsoft Windows Bitmap (BMP)* is a standard bitmap storage format used in the Windows environment, OS/2 (which is a strict superset of Windows), and some DOS-based programs. This format supports color depths of 1-bit, 4-bit, 8-bit, 16-bit, and 24-bit. Color depths of one and four bits correspond to monochrome and 16-color images, which you really shouldn't use in multimedia applications. 8-bit images contain a palette with the 256 colors that best describe the image. 24-bit images use True Color (see Chapter 7, *Manipulating Color and Pixels with Visual Basic*). In between palettes and True Color is another image quality called HiColor. HiColor images are comprised of 16 bits (five bits each for the green and blue components and six bits for red) and they contain 64,000 colors. For most practical purposes, they are good as True Color images.

The two basic types of bitmaps are:

- Device-dependent (BMP)
- Device-independent (DIB)

Device-dependent bitmaps (BMPs) are tied to the output device (the monitor or printer) because the bits of the bitmap and the pixels of the output display device are closely correlated. Device-independent bitmaps (DIBs), however, are not tied to a particular display device because they represent the appearance of the image. For purposes of this discussion, BMPs and DIBs are equivalent.

The JPEG Format

The *JPEG format (Joint Photographic Experts Group)* supports True Color and palette images and can achieve a higher degree of compression. It allows variable levels of compression and is *lossy* because it discards a little information in order to achieve a high degree of compression (more on lossy and lossless compression in a moment).

Graphics Interchange Format

CompuServe designed the *Graphics Interchange Format (GIF)* (pronounced “GIFF” or “JIFF”) mainly for a fast exchange of bitmapped images between computers. GIF files are relatively small, but displaying them is slow. This format is great for use on the Internet or electronic bulletin board systems. GIF is one of the most stable image file standards even though it only supports images that have 256 or fewer colors. Most of the images you’ll find on the World Wide Web are stored in GIF format.

TIP

Because GIF files are compressed by nature, running a file compression utility on them, such as WinZIP or PKZIP, does not result in smaller files.

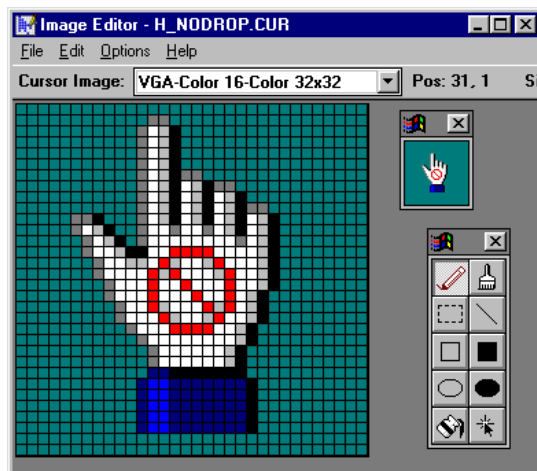
Microsoft Windows Icon Image

Microsoft Windows Icon Image (ICO) is the format of all Windows icons. Icons are usually stored as resources in a program. An icon image consists of a bitmapped image and a mask to make up the transparent parts of the picture. The mask is a color, usually the background color, that’s ignored when the image is rendered. Every pixel painted with the mask color lets the underlying bitmaps show through. An icon is typically 32×32 pixels and composed of 16 colors. When you minimize an application, its icon appears on the screen. A smaller version is displayed on the task bar in Windows 95/98.

To create or edit icons, you use an icon editor. *ImagEdit* is the icon editor that comes with the Professional Edition of Visual Basic 6. For an example of what you can do with it, see Figure C.2. You can also use *IconMagic*, an adequate shareware program for creating or editing icons.

FIGURE C.2:

Creating cursors (and icons)
with the ImagEdit utility



Microsoft Windows Cursor Image

Microsoft Windows Cursor Image (CUR) is the format of all Windows cursors. CUR is a special bitmap that indicates the current position of the mouse. You can define your own custom cursors or use any of the predefined Windows cursors. You can also use *ImagEdit* to edit cursors. Cursors are important in giving users feedback about the status of your application. For example, if your program is busy with a task, display a wait cursor such as the hourglass. Then, when the operation is complete, reset the hourglass to the standard cursor. Whichever cursors you decide to use, keep them simple and intuitive for the user.

In Visual Basic, you can use custom cursors to set the mouse pointer. By creating your own cursors, you can make your application unique and also show the current state of the user's actions in a more meaningful fashion. Visual Basic uses the *MousePointer* property to set the cursor and to load icons. When loaded, the icons are converted to the CUR format.

NOTE

An animated cursor is a special type of cursor. It consists of sequences of static cursors that are played back in rapid succession to produce a small animation. However, Visual Basic does not support animated cursors or color cursors.

To load a custom cursor or icon, you must use both the *MousePointer* and the *MouseIcon* properties. Follow these steps:

1. Set the *MousePointer* property to **99–Custom**.
2. Load the icon into the *MouseIcon* property or add the following statement to your Form:

```
Form1.MouseIcon = LoadPicture("c:\vb\graphics\cursors\wait08.cur")
```

Windows Metafile

The *Microsoft Windows Metafile (WMF)* format is used for storing vector data. Visual Basic doesn't provide any methods for storing information in metafiles, but you can still assign them to the *Picture* property of the controls that can display images.

Table C.2 summarizes the common image formats discussed so far.

TABLE C.2: Common Windows Image Formats

FILE FORMAT	FILE TYPE	EXTENSION
Windows Bitmap	Bitmap	BMP, DIB
JPEG	Bitmap	JPG
Graphics Interchange Format	Bitmap	GIF
Windows Icon	Bitmap	ICO
Windows Cursor	Bitmap	CUR
Windows Metafile	Vector	WMF

Image Compression

Compression involves making files smaller by removing repetitive patterns of data so that they can be stored in fewer bits and take up less disk space. Because graphics data usually require a lot of disk space, compression is important. In fact, regardless of the platform on which your application is running, you must use some compression method. You need to fit graphics data on the disk and sometimes in a block of memory. Your speed and compression ratio will depend on what you intend to do. In developing today's applications, you must be familiar with the major types of compression and how they are used.

TIP

When you assign an image to a Form or a PictureBox control, the bitmap is stored along with the Form in the corresponding FRX file, as explained in Chapter 2, *Visual Basic Projects*. Visual Basic uses the original file format to store the bitmap. If it's a BMP file, it is stored as a BMP bitmap. If the original file is in JPEG format, Visual Basic stores the bitmap in JPEG format. Therefore, you should try to assign compressed image formats to the Picture property at design time.

Types of Compression

The term *codec* is shorthand for *compressor/decompressor* and refers to the program that compresses and decompresses an image (or other types of information). Data that's not compressed is sometimes referred to as raw data or unencoded data. The term *compression ratio* refers to the ratio of uncompressed data to compressed data and is shown as 2:1, for example. In this case, the amount of uncompressed data is twice the amount of compressed data. The higher the ratio, the better the

compression scheme. Notice that the compression ratio refers to the amount of bytes required to store the information, not to the information itself. The two types of compression methods are lossy and lossless.

Lossy Compression This method discards any data that the compression method decides is not needed, and original data is lost when the file is decompressed. Although lossy compression methods achieve better ratios than lossless methods, their use depends on the type of image your application uses. For example, you would not want to use the lossy method for program files or medical X-rays, where the lost data is important. The lossy method is an excellent choice for graphics on a Web page or in printed media. In general, the information lost through lossy compression becomes evident only when the image is enlarged. If the image isn't going to be enlarged or processed in other ways, any artifacts introduced by lossy compression usually go unnoticed by the human eye.

Lossless Compression This method re-records a file's data in a more compact fashion. No data is lost when the file is decompressed. Let's assume you have a picture of the ocean. A large portion of the image consists of blue tones and a number of adjacent pixels have identical color. Instead of repeating the same Color value, say, 12 times, you can replace these 12 values with the number 12 followed by the Color value. When the decompressor sees this, it generates 12 identical Color values. Another compression trick is to take the difference of adjacent pixels. On the average, the pixels in any given area of an image have similar values. Instead of storing their absolute values, you can store their differences, which have small sizes and can be represented with fewer bits than the original Color values.

Although compression might not be an issue in business applications, it plays an important role in Web pages for the Internet or an intranet, on which large images translate into prolonged download times.

VB6 at Work: The Wipes Project

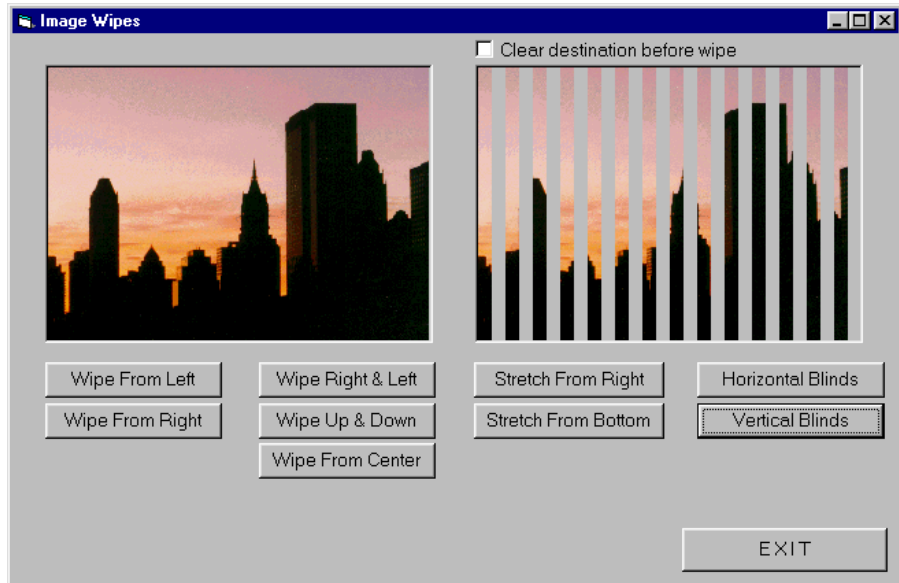
All the information you need to use images effectively from within your Visual Basic applications is covered in Chapter 6, *Drawing with Visual Basic*. One method that was not covered earlier is how to display a number of images in a sequence, like in a slide show. Because computer-controlled slide shows are popular, I developed the Wipes application, which demonstrates the basic concepts.

The Wipes application, shown in Figure C.3, displays an image on a PictureBox control with a smooth transition, or *wipe*. It can bring the image from the left or display it in stripes starting from the middle. Also included is a special type of wipe that stretches the image outside its destination and progressively resets it to its original size. The program uses a single image to demonstrate the various transitions,

and in your own application, you can probably keep the PictureBox control with the source image hidden. Each time you want to wipe another image, you must load it in the hidden PictureBox and activate the code of one of the buttons or similar code that performs another type of wipe.

FIGURE C.3:

The Wipes application demonstrates how to use the PaintPicture method to set up a slide show on your computer.

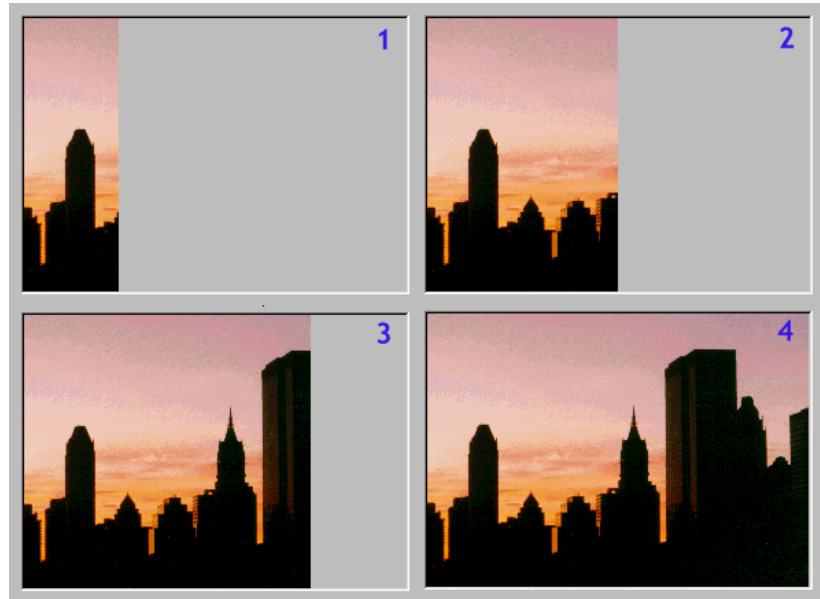


The transitions are achieved by moving blocks of pixels from one PictureBox control to the other with the PaintPicture method. The techniques are simple and you can come up with numerous transitions of your own. The basic idea is to copy blocks of pixels from the source image to the destination. By controlling the shape of the block and its origin and destination, you can create all types of transitions.

Right or Left Wipes Let's look at the wipe that progressively displays an image from left to right. The code behind the Wipe from Left button copies a vertical stripe from the source to the destination image and increases the stripe's width at each step. The first time, it copies a stripe that's one pixel wide that extends from the top to the bottom of the PictureBox control. It then increases the width of the stripe, copies it again, and repeats the process until the entire destination is covered. Figure C.4 shows how this works.

FIGURE C.4:

A few stages of the Wipe from Left wipe

**Code C.3:****The Wipe from Left Button's Click Event Handler**

```
Private Sub WipeLeft_Click()
    Dim X

    If ClearDestination.Value Then Picture2.Picture = LoadPicture()
    For X = 1 To Picture1.ScaleWidth
        Picture2.PaintPicture Picture1.Picture, 0, 0, _
            X, Picture1.ScaleHeight, 0, 0, X, _
            Picture1.ScaleHeight, &HCC0020
    Next
End Sub
```

In Code C.3, *X* is the width of the stripe being copied. Its initial value is 1 (a one-pixel-wide stripe), which increases by an increment of 1 until the value equals the control's width. *ClearDestination* is the CheckBox above the destination PictureBox that lets the user specify whether the destination PictureBox will be cleared before a new transition or not.

Stretch Wipes The Stretch wipes also copy a stripe of the original image, except the destination's width and height are not the same as the source's dimensions. Instead, the stripe is stretched to fill the entire destination. In the first step, the

destination image is filled with the first column of pixels of the source image. The destination is then filled with the first two columns of pixels, and so on, until the entire source image is copied onto the destination. Following is the code of the Stretch from Right and Stretch from Left wipes.

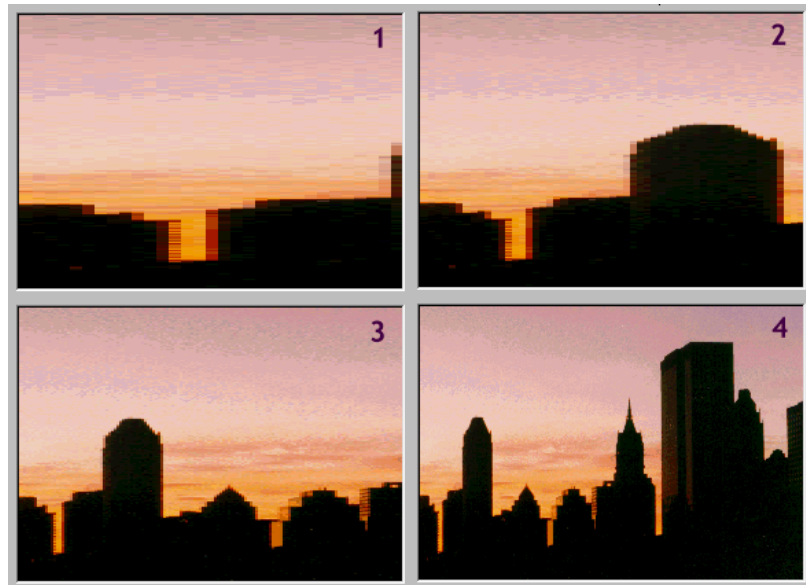
Code C.4: The Stretch from Right Wipe

```
Private Sub StretchRight_Click()  
Dim X  
  
If ClearDestination.Value Then Picture2.Picture = LoadPicture()  
For X = 1 To Picture1.ScaleWidth Step 3  
    Picture2.PaintPicture Picture1.Picture, 0, 0, _  
        Picture1.ScaleWidth, Picture1.ScaleHeight, 0, 0, X, _  
        Picture1.ScaleHeight, &HCC0020  
Next  
  
End Sub
```

Figure C.5 shows how the Stretch from Right effect works. During the first stages, a small vertical stripe of the original image covers the entire destination picture box. This small stripe becomes increasingly wider, until the entire source image is copied to the destination picture box.

FIGURE C.5:

A few stages of the Stretch from Right wipe



Code C.5: The Stretch from Bottom Wipe

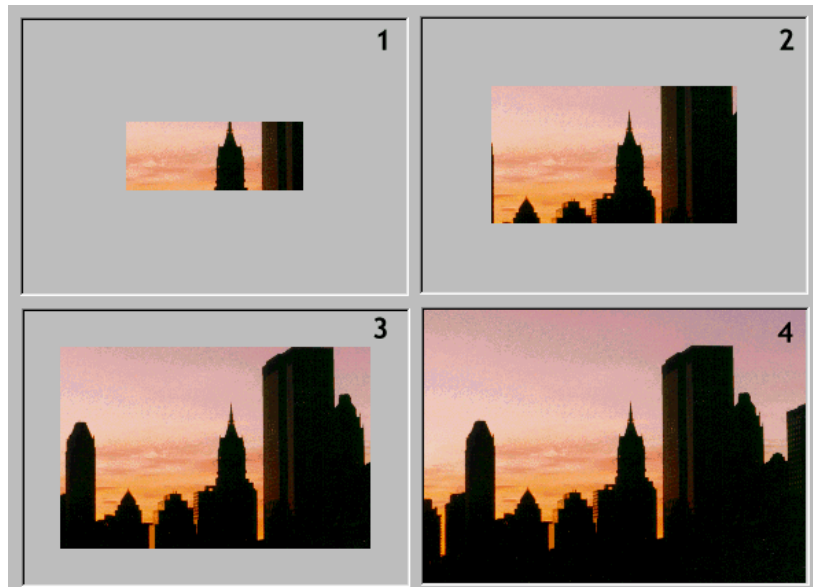
```
Private Sub StretchBottom_Click()  
Dim X  
  
If ClearDestination.Value Then Picture2.Picture = LoadPicture()  
For X = 1 To Picture1.ScaleHeight Step 3  
    Picture2.PaintPicture Picture1.Picture, 0, 0, _  
        Picture1.ScaleWidth, Picture1.ScaleHeight, 0, 0, _  
        Picture1.ScaleWidth, X, &HCC0020  
Next  
  
End Sub
```

The OpCode value used with the PaintPicture method (the method's last argument) has the default value and could be omitted. You can try different values to create additional special effects with the Wipes application.

Center Wipes To understand how the Wipe from Center effect works (see Figure C.6), imagine that the two images are square. The wipe starts by copying the middle pixel, then it copies a 3×3 block centered around the pixel at the middle of the source image, then it copies a 5×5 block, and so on, until the destination is filled.

FIGURE C.6:

A few stages of the Wipe from Center effect



Because most images aren't square, the first block isn't a single pixel, but a row or column of pixels. The number of pixels is chosen so that after the initial block is copied, the following blocks are two pixels wider and taller than the previous one, and they approach all four edges at the same speed.

Code C.6: The Wipe from Center Effect

```
Private Sub WipeCenter_Click()
Dim PWidth, PHeight As Integer
Dim i As Integer

If ClearDestination.Value Then Picture2.Picture = LoadPicture()
If Picture1.ScaleWidth > Picture1.ScaleHeight Then
    PWidth = Picture1.ScaleWidth - Picture1.ScaleHeight
    PHeight = 1
ElseIf Picture1.ScaleWidth < Picture1.ScaleHeight Then
    PWidth = 1
    PHeight = Picture1.ScaleHeight - Picture1.ScaleWidth
Else
    PWidth = 1
    PHeight = 1
End If

For i = 1 To Picture1.ScaleWidth - PWidth
    Picture2.PaintPicture Picture1.Picture, _
        Int((Picture1.ScaleWidth - PWidth) / 2), _
        Int((Picture1.ScaleHeight - PHeight) / 2), _
        PWidth, PHeight, _
        Int((Picture1.ScaleWidth - PWidth) / 2), _
        Int((Picture1.ScaleHeight - PHeight) / 2), _
        PWidth, PHeight, &HCC0020
    PWidth = PWidth + 1
    PHeight = PHeight + 1
Next

End Sub
```

Blinds The wipes that use blinds are a bit more complicated. The Horizontal Blinds wipe draws multiple stripes of equal size at equal distances from each

other. With each iteration, the stripes get taller, until they touch each other. The program starts by calculating the number of stripes required to cover the entire destination. Each stripe is one pixel tall and grows to *StripeHeight* pixels. Therefore, the number of stripes required to cover the destination is stated as follows:

```
StripeHeight = 20  
Stripes = Fix(Picture1.ScaleHeight / StripeHeight)
```

Then a double loop begins. The outer loop is repeated as many times as the maximum number of pixels in each stripe (which is given by the *StripeHeight* variable). The inner loop copies stripes of pixels and is repeated *Stripes* times. With each iteration, the stripe's width increases by one pixel.

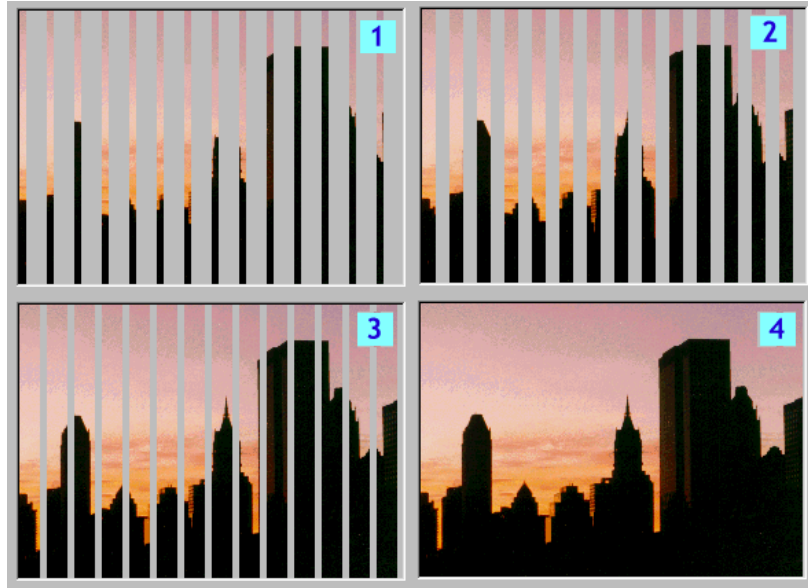
Code C.7: The Horizontal Stripes Effect

```
Private Sub Horizontal_Click()  
Dim Stripes As Integer  
Dim i, j As Integer  
Dim StripeHeight As Integer  
  
If ClearDestination.Value Then Picture2.Picture = LoadPicture()  
StripeHeight = 20  
Stripes = Fix(Picture1.ScaleHeight / StripeHeight)  
On Error Resume Next  
For j = 1 To StripeHeight  
    For i = 0 To Stripes  
        Picture2.PaintPicture Picture1.Picture, 0, i * StripeHeight, _  
            Picture1.ScaleWidth, j, 0, i * StripeHeight, _  
            Picture1.ScaleWidth, j, &HCC0020  
    Next  
Next  
  
End Sub
```

Figure C.7 shows how the Vertical Blinds special effect works. During the first stages, a few thin vertical stripes are copied from the source to the destination picture box. These stripes become increasingly wider, until they cover the entire destination picture box.

FIGURE C.7:

A few stages of the Vertical Blinds wipe



Timing the Transitions

The speed of each wipe depends on the speed of the host computer. If you run the Wipes application on a Pentium™ 400, it will complete in no time at all and you may not even notice it. Ideally, each wipe should take a second or two to complete, and you should be able to specify the duration from within your code. To time the transitions, you must add some code that introduces small delays between successive stages of a transition. Let's say a specific transition must copy 100 bands of pixels and it must finish in two seconds. Each band must take $2,000/100$ or 20 milliseconds. If the computer can transfer the band in 20 milliseconds or less, then you simply can't make the transition fast enough. There's nothing you can do, short of letting the computer do its best.

If the computer is fast enough to transfer the band in less than 20 milliseconds, say 15 milliseconds, you must add some code that will cause the transition to delay for 5 more milliseconds before it starts transferring the next band. This way, the entire transition will complete in two seconds, no matter how fast the host computer is. In short, you must divide the total duration of the transition by the number of bands of pixels to be transferred. After each band has been transferred, you must check whether there's any time left. If so, loop locally to make up for the computer's speed. If not, the program should proceed with the next band.

In the TWIPES folder under the WIPES folder, you'll find the timed version of the WIPES project. In the Form's Load event, I've added the following statement:

```
TotalDuration = 2000
```

TotalDuration is a Form variable that holds the wipe's duration in milliseconds. Let's see how this variable is used by the Right to Left transition. The revised code of the WipeRight button is shown next:

```
Private Sub WipeRight_Click()
    Dim X As Integer
    Dim mseconds As Integer

    If ClearDestination.Value Then Picture2.Picture = LoadPicture()
    mseconds = TotalDuration / Picture1.ScaleWidth
    For X = 1 To Picture1.ScaleWidth
        StartDelay
        Picture2.PaintPicture Picture1.Picture, _
            Picture1.ScaleWidth - X, 0, _
            X, Picture1.ScaleHeight, _
            Picture1.ScaleWidth - X, 0, _
            X, Picture1.ScaleHeight, &HCC0020
        EndDelay mseconds
    Next
End Sub
```

The variable *mseconds* is the number of milliseconds between the transfers of successive bands, and it's the total duration of the wipe divided by the number of bands to be transferred. Since the Right to Left transition transfers one vertical line of pixels at a time, the *mseconds* variable's value is *TotalDuration*/*Picture1.ScaleWidth*.

Each time a new band of pixels is transferred, the program calls the *StartDelay()* subroutine, which marks the exact current time. Here's the *StartDelay()* subroutine's listing:

```
Sub StartDelay()
    StartTime = timeGetTime()
End Sub
```

I'm using the *timeGetTime()* API function (see Chapter 13 for more information on API functions), which has a better resolution than the *Timer()* function. The *Timer()* function can't reliably measure intervals under 55 milliseconds, which is too large for the needs of the WIPES application. The *timeGetTime()* function is accurate down to a millisecond, making it just right for this application.

After the first band has been transferred to the target PictureBox, the *EndDelay()* subroutine is called. This subroutine is called with an argument, which is the duration of the current band (*mseconds*). The *EndDelay()* subroutine gets into a loop and

doesn't return before *mseconds* milliseconds have elapsed since the `StartDelay()` subroutine was called. Here's the definition of the `EndDelay()` subroutine:

```
Sub EndDelay(N As Integer)
    While timeGetTime() - StartTime < N
    Wend
End Sub
```

Open the WIPES project in the TWIPES folder on the CD and examine the code of the other transitions. They all use the same logic: They divide the total duration of the transitions among the bands to be transferred and then make sure that each band transition takes (at least) as many milliseconds to complete. On a very fast computer, most of the application's execution time will be spent in the loop of the `EndDelay()` subroutine. Should you test this application with the next generation of Pentium™ processors, the wipes will probably be abrupt. Each band will be transferred instantly and then you'll have to wait for the next one. You may even have to introduce additional delays between individual pixel transfers and band transfers.

Enhancing Applications with Sound

Besides including great-looking pictures, you can enhance an application with sounds. You can add two types of sounds to your applications:

- Recorded sounds (files with the extension WAV)
- Synthesized, or MIDI, sounds (files with the extensions MID)

You can also control the CD player to play back tracks from an audio CD, but this technique is used almost exclusively with programs (mostly games) that require the presence of a CD in the CD-ROM drive.

TIP

Many developers will consider adding sound capabilities to their application after everything else is in place. If you want to make sounds an integral part of your application, incorporate them early in the development cycle and find ways to make them part of the application's user interface. Later in this appendix, you'll learn how to use sound to provide audio messages to the users of your applications.

Using Waveform Audio

This is the format used for digitized sounds, which are stored in files that have the extension WAV. If you have a sound card and a microphone, you can record your own sounds. You can also use the WAV files included on the CD. On it you'll find

some useful sound files, such as the names of the days and months, numbers, and more. In the following sections, you'll see how to use sound files to add audio features to your applications.

Sound quality depends on two major factors:

- The sampling rate
- The number of bytes allocated to each sample

The *sampling rate* is the number of audio samples taken per second. The sampling rate varies from 8KHz (8,000 samples per second) to more than 40KHz (40,000 samples per second). The more samples, the more accurately you can later reconstruct the original sound, but also the more disk space the file consumes.

Lower sampling rates can only be used with voice. Sounds taken at 8,000 samples per second are equivalent to telephone quality. You can recognize the speaker, but that's about it. A sampling rate of 11,025 samples per second is a good trade-off between storage requirements and quality, but it's still low quality. The most common sampling rate for digitizing human speech is 22,050 samples per second (known as radio quality).

The highest sampling rate you can achieve with a sound card (and play back on a sound card) is 44,100 samples per second. This sampling rate yields the best possible sound quality and is known as CD quality. As you can easily calculate, sounds taken at this sampling rate consume four times as much disk space as telephone quality sounds.

The second factor that determines quality is the number of bytes allocated to each sample. Each sample represents the intensity of the sound at any given instance in time. Sound samples are represented as integers in the computer's memory. If you allocate one byte to each sample, there are 256 levels of intensity. The smallest value, zero, corresponds to silence, and the largest value, 255, corresponds to the sound's maximum intensity. If you use an integer to represent each sample, you have more than 30,000 levels of intensity (you can represent 32,768 different values with two bytes). Thus, the samples are represented more accurately in the computer's memory. As you can also easily calculate, this sound will take twice as much disk space as a sample represented by one byte.

TIP

There are remarkable similarities between sounds and images. For example, a sound's sampling rate corresponds to an image's resolution and the sampling accuracy corresponds to the number of colors. Raising the sampling rate is equivalent to scanning an image with more dots per inch. Raising the sampling accuracy (representing a sample with an integer instead of a byte) is equivalent to raising an image's color resolution (e.g., True Color instead of a 256-color palette).

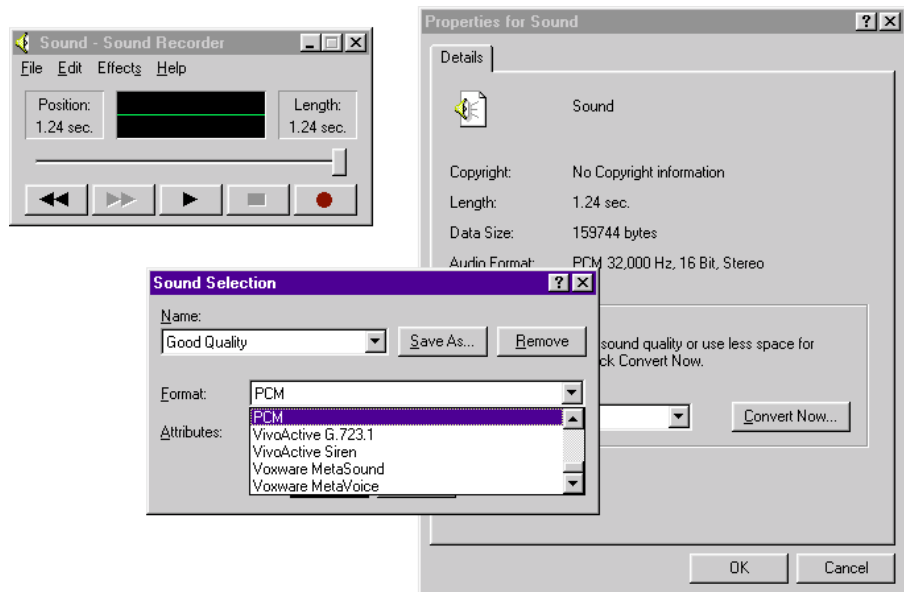
One other factor that affects the quality of digitized sound is whether the sound is digitized in mono or stereo. A stereo recording has two separate channels, which means twice the number of samples (and twice the disk space) of a mono recording.

Sounds also can be compressed. If you surf the Web, you're probably familiar with the various sound compression schemes, such as RealAudio. Even Windows 95 includes codecs for sound files. One of them, the TrueSpeech codec, can squeeze sounds sampled with one byte per sample down to one bit per sample. To see the codecs installed on your system, follow these steps:

1. Choose Start > Programs > Accessories > Multimedia > Sound Recorder.
2. In the Sound Recorder dialog box, choose File > Open, and select a WAV file.
3. Choose File > Properties to open the Properties for Sound dialog box, as shown in Figure C.8.

FIGURE C.8:

The sound codecs can be located in the Sound Selection window of the Properties for Sound dialog box.



4. Click the Convert button to open the Sound Selection window.
5. Click the Format drop-down list arrow to see a list of the codecs installed on your system.

In general, computers can handle sound files easily, and there's no need to compress sound files unless you're going to place them on the Web or you have many large files to distribute with your application.

Using MIDI

MIDI (pronounced “MIDDY”) is an acronym for *Musical Instrument Digital Interface*, a serial interface standard that can connect music synthesizers, musical instruments, and computers. A MIDI file doesn’t contain the sound, it contains notes and durations that describe a tune and tell the computer how a sound can be constructed. With the help of the appropriate hardware, these instructions are translated into sounds.

As described earlier, a WAV file stores digitized sound. A WAV file is a pictorial representation of the original sounds. You can think of the WAV file as the image of a printed page (much like a fax), and you can think of the MIDI file as the textual information that corresponds to the fax. The pictorial representation of the page may take 100 times as much disk space as the text file with the fax’s contents. Using MIDI files reduces storage requirements to a minimum.

To record sound files (and edit them to some extent), you can use the Sound Recorder application that comes with Windows. For more advanced sound processing tools, you can resort to a couple of excellent shareware applications such as CoolEdit, which is available from:

<ftp:syntrillium.com>

and GoldWave, which is available from:

<http://web.cs.mun.ca/~chris3/goldwave/goldwave.html>

Using the MCI Control

The *Media Control Interface*, or *MCI*, was introduced as part of the Multimedia Extensions provided with Windows 3.x. MCI is now part of the Windows operating system and one of several high-level multimedia services provided by it. You can easily access these services through Visual Basic controls.

MCI is designed to communicate with a plethora of device types, including waveform audio, MIDI, audio CD, and MPEG video. It does so by communicating with device drivers, which interpret MCI commands into the device’s command set, thus driving the device. MCI can communicate with devices in two ways:

- By sending text commands to the device driver (through the string interface)
- By sending commands directly to the devices

Because the string interface is much easier to use, we will limit our discussion to it.

About MCI Device Types

MCI device types can be simple or compound. *Simple MCI devices* do not require a data file for playback. Videodisc players and audio CD players are simple device types. These devices are called simple because they stream the data from an external source to an output. The computer is hardly involved in the process.

Compound MCI devices require a data file for playback and the computer is involved in the process. MIDI sequencers and waveform audio players are compound device types. To play back an audio file, for instance, the samples must be decompressed and then fed to the audio card with the appropriate timing information. Table C.3 lists all the MCI devices that respond to the common set of commands.

TABLE C.3: MCI Device Types

DEVICE TYPE	DESCRIPTION
animation	Animation device
cdaudio	Audio CD player
dat	Digital audio tape player
digitalvideo	Digital video in a window
waveaudio	Audio device that plays digitized waveform files
videodisc	Videodisc player
vcr	Videotape recorder or player
sequencer	MIDI sequencer
scanner	Image scanner
overlay	Overlay device
other	Undefined MCI device

MCI devices recognize plain English commands, such as play, stop, and resume. To send a command to an MCI device, use the `mciSendString()` function. The `mciSendString()` function passes an MCI command to the MCI device, which executes the command and reports back a return code. If all goes well, the return code is zero. If not, the return code is a number that indicates the error that occurred.

The error code isn't very helpful; however, it's possible to retrieve the description of the error by passing the code to the `mciGetErrorString()` function. This function will return an error description such as "The command was carried out" or "The file cannot be played back on the specified MCI device."

The basic syntax of all MCI commands contains a verb, an object, and a modifier. The verb describes the action to take place, such as “play” or “stop.” The object is the device on which the verb acts. For example, to play back the file *Months.wav*, you would issue the following MCI commands:

```
open months.wav type waveaudio alias months
play months
close months
```

This command sequence requests that the *Months.wav* file be opened on the *waveaudio* device with the alias *months*. The file is then played back and finally closed. When the file is closed, the device is free to play back another file. In the second MCI command, *play* is the verb and *months* is the object.

The modifier (if any) determines how the waveform will be played back. For example, you can request that only a section of the sound file be played back. If you want to play back the name of the first month only, you can use the *from* and *to* modifiers:

```
play months from 0 to 1200
```

This command plays back only the first 1200 samples of the *months* file, which presumably correspond to the spoken word “January.” The parameters *from* and *to* are the command’s modifiers (they modify the default behavior of the command).

Executing MCI Commands

The MCI commands are simple to understand and use. They are actually simpler than Visual Basic’s commands. But passing these commands to the MCI interface isn’t quite as simple. To issue an MCI command, you must use the *mciSendString()* function, an API function (see Chapter 13, *The Windows API*, for a detailed discussion). To use this function in a Visual Basic application, you first declare it as follows:

```
Declare Function mciSendStringA Lib "MMSystem" _
    (ByVal mciCommand As String, ByVal returnString As String, _
    ByVal returnLength As Integer, _
    ByVal callBack As Integer ) As Long
```

NOTE

The suffix A at the end of an API function name is a convention introduced with Win32 to differentiate between 16-bit and 32-bit functions. For more information, see the introduction in Chapter 13.

The last argument in the declaration will always be zero because you can’t use callback functions with Visual Basic. The *mciCommand* argument is a string that holds the MCI command (the string “play”, for instance). The *returnString* argument

holds a string with a response from the MCI device. Some MCI commands request information from the MCI interface, such as the characteristics of an audio file or the current track on an audio CD or an AVI file. This information is returned via the *returnString* argument. The *returnLength* argument is the length of the *returnString* argument. The value returned by the `mciSendString()` function is a Long Integer, indicating the success of the operation (if it's zero) or the reason for its failure (if it's positive).

To use the `mciSendString()` function to send commands to an MCI device, follow these steps:

1. Declare the `mciSendString()` function.
2. Declare the argument *returnString* as `String * 255`.
3. Call the `mciSendString()` function.

```
Dim errorCode As Integer
Dim returnStr As String * 256
Dim returnCode As Integer
```

```
errorCode = mciSendStringA("play months", returnStr, 255, 0)
```

You must examine the code returned by the function, which indicates the success or failure of the call. You should actually pass this value to the `mciGetErrorString()` function, which will translate this number to an actual error message. We will discuss the `mciGetErrorString()` function shortly, but let's first look at the MCI commands you can use to control the MCI devices.

Using the MCITest Application

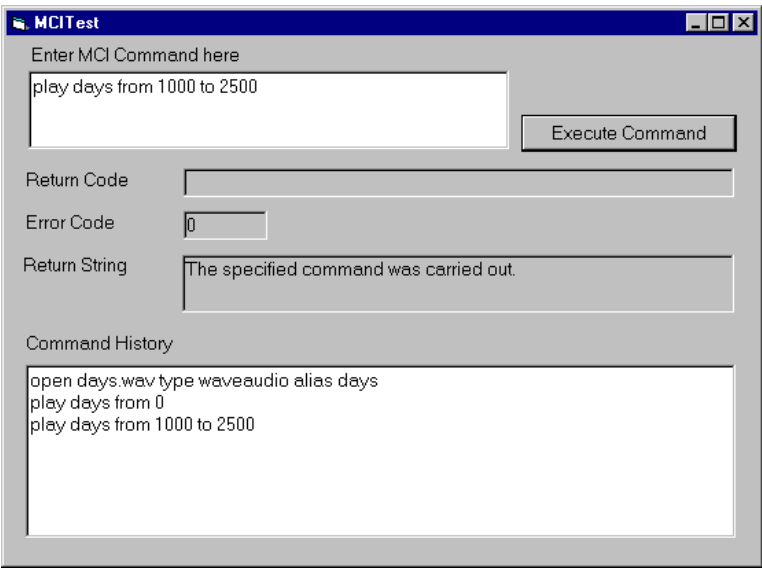
As you read the following sections, you can experiment with the MCITest application, shown in Figure C.9, which lets you type plain MCI commands, passes them to the MCI interface, and displays the string returned by the MCI interface.

Enter the MCI commands you want to pass to the MCI interface in the Enter MCI Command Here textbox and then click the Execute Command button. The MCI command is executed and the result of the operation is displayed in the Return Code, Error Code, and Return String boxes. Error Code is the value returned by the `mciSendString()` function, and Return Code is the error message that corresponds to this value. Return String is the value of the `mciSendString()` function's *returnString* argument.

If the MCI command executes successfully, it is added in a separate line in the Command History textbox at the bottom of the Form. You can select any command from the Command History textbox with the mouse, copy it, and then paste it in the Enter MCI Command Here textbox.

FIGURE C.9:

The MCITest application lets you issue MCI commands directly to the various MCI devices.



Now let’s look at the various MCI commands for controlling the MCI devices. Their number is small, but most accept many arguments, and their functions differ depending on the device to which they apply. The *open* command, for example, opens any MCI device, but it accepts different arguments, depending on whether it’s used to open an MIDI file or animation device. Table C.4 summarizes the most commonly used MCI commands.

TABLE C.4: Commonly Used MCI Command Strings

COMMAND	DESCRIPTION
capability	Requests information about the capabilities of a device
close	Closes a device
info	Requests information about a device
open	Opens and initializes a device for use
pause	Pauses playing or recording on a device
play	Begins playing on a device
record	Begins recording on a device
resume	Resumes playing or recording on a paused device

Continued on next page

TABLE C.4 CONTINUED: Commonly Used MCI Command Strings

COMMAND	DESCRIPTION
seek	Changes the current position in the media
set	Changes control settings on the device
status	Requests information about the status of a device
stop	Stops playing or recording on a device

Opening a Device

Before you can use a device, you must initialize it by using the open command. The open command loads the driver into memory and sets a device ID for future use with other MCI commands. The amount of memory dictates the number of devices that you can have open. The open command has the following arguments:

- **alias** *alias* An alias by which you can refer to the device
- **shareable** Allows applications to share a common device
- **type** *devicename* Identifies the MCI device name when device refers to a media element instead of to the MCI device name

The following sample command string opens the audio CD device and sets an alias for it:

```
open cdaudio alias CD
```

The commands that follow may refer to this device by its alias.

If you're opening a compound device, you must also specify the file that will be associated with the device, for instance:

```
open c:\vb\apps\audio\Months.wav type waveaudio alias months
```

The *Months.wav* file is opened as *months*, and its alias is used to refer to it in subsequent MCI commands. For example, to play back the sound file, you would use the following command:

```
play months
```

Retrieving Information about Devices

You can get information from an open device by using the *capability*, *status*, and *info* commands. For example, the following string determines if the audio CD player can play:

```
capability cdaudio can play
```

If you pass this command string to an MCI device, it will return True if the audio CD device can play.

The capability Command The arguments of the capability command let you find out the capabilities of a specific device. For example, to find out if an open device can play backward, use the argument *can reverse*. The MCI interface's response is returned via the *returnString* argument of the *mciSendString()* function. If you issue the following command with the MCITest application, the response will be False:

```
capability cd has video
```

You can use the following arguments with the capability command:

<i>can eject</i>	<i>can play</i>	<i>can record</i>
<i>can reverse</i>	<i>can save</i>	<i>compound device</i>
<i>device type</i>	<i>has audio</i>	<i>has video</i>
<i>inputs</i>	<i>outputs</i>	<i>uses files</i>
<i>uses palettes</i>		

The info Command The info command returns information about the currently selected MCI device. You can use the following arguments with the info command:

- **input** The device input (for the waveaudio device, it will be Microsoft Sound Mapper)
- **output** The device output
- **file** The file associated with the device (applies to compound devices only)
- **product** The name of the hardware device that corresponds to the MCI device

The status Command The status command returns information about the status of the currently selected device. The type of status information you can receive depends on the device. Following are some common arguments for the status command:

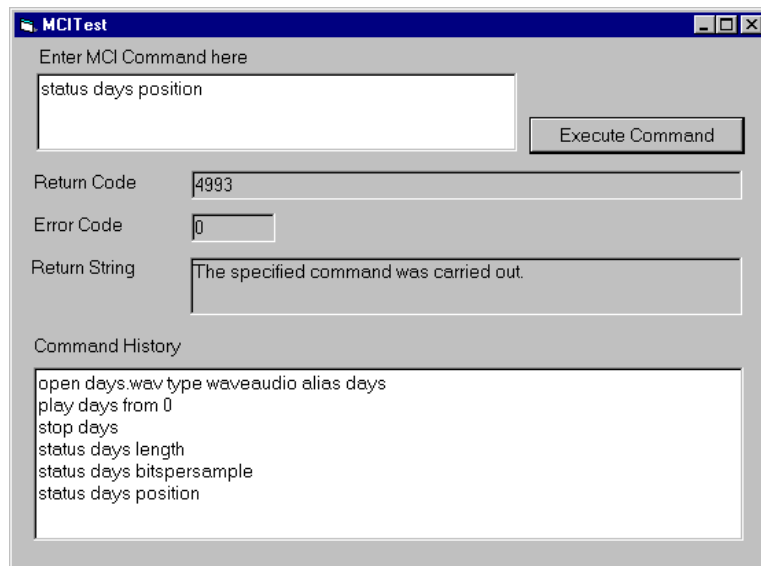
- **channels** Number of channels in an audio device
- **bitspersampl** Bits per sample for a waveaudio device
- **mode** Returns the “not ready”, “paused”, “playing”, “seeking”, or “stopped” string, depending on the current activity of the device

- **position** Returns the current position in the open file. The position is reported in the current time format (see the discussion of the set command later in this chapter).
- **ready** Returns True if the device is ready
- **length** Returns the total number of frames in an animation or the total number of samples in a WAV file
- **window handle** Returns the handle of the window that's used for animation playback

In the MCITest application, the “status days position” MCI string requests the current location in the Days.wav file (which was opened with the alias *days*, as you can see in the history list). This location is reported in the Return Code box (see Figure C.10).

FIGURE C.10:

If the MCI command requests information about a device, the response is displayed in the Return Code box.



Playing a File

Once a device is open, playback can begin with the *play* command. If no arguments are set, the device plays until the command is stopped or until the end of the file is reached. The following command starts playback on the audio CD device:

```
play cdaudio
```

You can use three arguments with the `play` command:

- **from** Specifies the start location of the segment to be played
- **to** Specifies the end location of the segment to be played
- **wait** Specifies that the computer should wait for the playback to complete before executing another command.

The file `Days.wav`, for instance, contains the names of the days of the week. To play back the name of a specific day, you would use the following command:

```
play days from 0 to 1000
```

This command plays the first 1000 milliseconds (one second) from the beginning of the file.

When you use the *wait* argument, the MCI device waits for the playback to end before relinquishing control to the application. If you don't specify the wait argument, another play command will interrupt the file being played. The following statement plays the open file with the alias *days* and waits until the playback is complete:

```
play days wait
```

The wait argument ensures that the entire sound will be heard.

Pausing and Stopping a Device

You pause playback on an MCI device with the *pause* command and stop playback with the *stop* command. To start a paused device again, you use the *resume* command. To start a stopped device, you use the *play* command. Issue the following commands with the MCITest application to see how the pause, stop, and resume commands behave:

```
play cdaudio  
pause cdaudio  
resume cdaudio  
stop cdaudio  
play cdaudio
```

Setting Device Properties

You use the *set* command to establish various properties of MCI devices. The form of the set command is similar to that of the capabilities command. The device name and an argument follow the name of the command. You can use the following

arguments with the `set` command for all compound devices (some arguments may not apply to all devices):

- **audio all off** Turns off the audio
- **audio left off** Turns off the left audio channel
- **audio right off** Turns off the right audio channel
- **video off** Turns off the video playback on an animation device
- **audio all on** Turns on both audio channels
- **audio left on** Turns on the left audio channel
- **audio right on** Turns on the right audio channel
- **video on** Turns on the video playback on an animation device
- **time format** Returns or sets the measurement unit for the file

The most important argument of the `set` command is the *time format* argument, which sets the measurement format of a file (if it's a compound device) or the medium (if it's a simple device). The value of the time format depends on the device. For waveaudio devices, the time format can be milliseconds, bytes, or samples. For animation devices, the time format can be either milliseconds or frames. The following MCI command sets the time format for the file *months* (opened with the waveaudio device) to *samples*:

```
set months time format samples
```

If you use the *seek* command (see the following section for details about the *seek* command) to move the current pointer in the file, you must specify the new location of the samples:

```
seek months to 3000
```

Likewise, if you request the file's total length with the following command, the length is reported in samples:

```
status months length
```

Audio CD devices use the *tms format*, which stands for track, minutes, seconds. To select a tune on an audio CD, you specify the track number as follows:

```
seek cd to 3:00:00
```

This command moves the current pointer to the beginning of the third track. To skip part of the tune, use a statement like the following:

```
seek cd to 3:01:00
```

This positions the current pointer 60 seconds after the beginning of the third track.

Other Common Commands

Some other useful MCI commands are *cue*, *delete*, *record*, and *seek*, which give the programmer more control over the function of the various devices.

The cue Command This command sets up a waveform device so that it can be ready to play or record with minimal delay. You can use two arguments with the cue command: *input* (to record) and *output* (for playback). The following MCI command prepares the waveaudio device for recording:

```
cue waveaudio input
```

The delete Command This command deletes part of a waveform file, and you can use two arguments with it: *from a position* (first sample to be deleted) and *to a position* (last sample to be deleted). *Position* must be expressed in the current time format. To delete the current sound, use a statement like the following, in which *wave* is the alias of the device:

```
delete wave
```

The record Command Use this command to start recording on a device. It can take the following arguments: *insert*, *from*, *to*, or *overwrite*. The *insert* argument cause the device to insert the new samples at the current location in the file, the *from* and *to* arguments let you specify where the new sample will be inserted and the *overwrite* argument specifies whether existing samples can be overwritten.

The save Command Use this command to save a recorded file. The save command follows the record command, as shown here:

```
open waveaudio alias recwave
record recwave from 0 to 3000
save recwave c:\sounds\noise.wav
close recwave
```

Notice that the waveaudio device can be opened without a filename. In this example, the waveaudio device is opened for recording with the alias *recwave*. The file is specified later, when the recorded sound is saved on disk.

The seek Command To move to a new position in a waveform file, you use the seek command with the arguments *to*, *to start*, or *to end*. You seek a specific sample with the *to* argument or move to the two ends of the file with the other two arguments. If the time format for the waveaudio device is milliseconds, the following statements allow you to skip the first four seconds in the sound before recording and overwrite the samples that follow:

```
seek wave to 4000
record wave overwrite
```

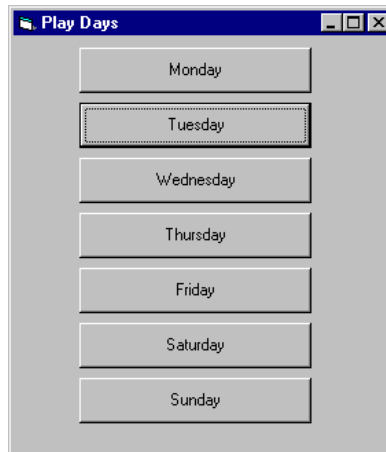
VB6 at Work: The PlayDays Project

PlayDays is a simple application that demonstrates the `mciSendString()` function call with the waveaudio device, which is the most common multimedia device.

The application uses the open and play commands to open and play the names of the days of the week that are stored in the `Days.wav` file. The MCI command that actually plays back the names is the play command with the *from* and *to* arguments. Be sure that `Days.wav` is in your current directory. To hear the name of a day, click its button, as shown in Figure C.11.

FIGURE C.11:

The PlayDays application



The code of the application is straightforward, except for figuring out the starting and ending locations of each day's name in the `Days.wav` file. For this, you must use a sound processing application that lets you select segments of the audio and play them back. CoolEdit is an excellent shareware application for processing sounds and producing new ones.

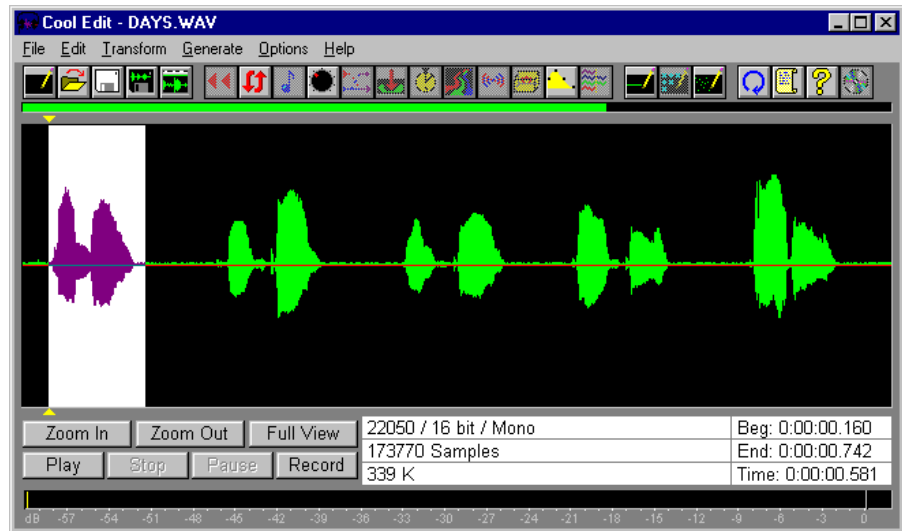
You can use CoolEdit, shown in Figure C.12, to select a section of an audio file and play it back. The selected sound segment in Figure C.12 corresponds to the word *Monday* in the file. In the lower-right corner of the window, the program reports the starting and ending locations of the selected segment in milliseconds. By default, the program reports these locations in samples, but you can click the corresponding boxes to see these values in milliseconds. (Or you can change the time format of the device to specify them in samples.)

In PlayDays, the word *Monday* starts at approximately 160 milliseconds and ends at 750 milliseconds. The MCI command to play back the word *Monday* is as follows:

```
play days.wav from 160 to 750
```

FIGURE C.12:

CoolEdit is a sound processing application that can be used to find the starting and ending locations of each word in a sound file.



You can easily spot the remaining words in the file and extract their starting and ending locations. Then, use these values with the *from* and *to* parameters of the play command. The complete listing of the PlayDays application follows.

Code C.8: The PlayDays Application

Option Explicit

```
Private Declare Function mciSendStringA Lib "WinMM" _
    (ByVal mciCommand As String, ByVal returnStr As String, _
    ByVal returnLength As Integer, _
    ByVal callBack As Integer) As Long
```

```
Private Sub Play_Click(Index As Integer)
```

```
    Dim errorCode As Integer
    Dim returnStr As String * 256
    Dim cmd As String * 255
```

```
    cmd = "open " & App.Path & "\days.wav type waveaudio alias days"
    errorCode = mciSendStringA(cmd, returnStr, 255, 0)
```

```
    If Index = 0 Then errorCode = mciSendStringA("play days from 150 _
        to 850 wait", returnStr, 255, 0)
```

```
    If Index = 1 Then errorCode = mciSendStringA("play days from _
        1200 to 1900 wait", returnStr, 255, 0)
```

```

    If Index = 2 Then errorCode = mciSendStringA("play days from _
        2200 to 2900 wait", returnStr, 255, 0)
    If Index = 3 Then errorCode = mciSendStringA("play days from _
        3100 to 4000 wait", returnStr, 255, 0)
    If Index = 4 Then errorCode = mciSendStringA("play days from _
        4300 to 4900 wait", returnStr, 255, 0)
    If Index = 5 Then errorCode = mciSendStringA("play days from _
        5100 to 5800 wait", returnStr, 255, 0)
    If Index = 6 Then errorCode = mciSendStringA("play days from _
        6100 to 6800 wait", returnStr, 255, 0)
End Sub

```

The mciGetErrorString() Function

If you run the PlayDays application and don't hear a sound, you won't be able to easily locate the mistake. It could be that you don't have a waveaudio device installed or that the waveaudio device didn't find the audio file. The `mciSendString()` function returns a code that indicates if the function has been completed successfully. If no error has occurred, the function returns a code of zero. If an error has occurred, you can use the `mciGetErrorString()` function to get information about the error code. The `mciGetErrorString()` function accepts the value returned by the `mciSendString()` function and returns a string that describes the error in English.

You can declare the 32-bit version of the function as follows:

```

Declare Function mciGetErrorStringA Lib "winmm" _ (ByVal errorCode _
    As Long, ByVal errorString As String, _
    ByVal returnLength As Integer) As Integer

```

The *errorCode* argument is the value returned by `mciSendString()`, *errorString* contains the error message on return, and *returnLength* is the length of the *errorString* variable.

Usually, you call the `mciSendString()` function to send a command to the MCI interface and then examine the return value with the `mciGetErrorString()` function, as shown next:

```

Dim returnStr, errorStr As String * 255
errorCode = mciSendStringA("play audio", returnStr, 255, 0)
returnCode = mciGetErrorStringA(errorCode, errorStr, 255)
MsgBox errorStr

```

VB6 at Work: The MCITest Project

Now, let's look at the code of the MCITest application (shown previously in Figures C.9 and C.10), which is quite straightforward. Each time the user clicks on the Execute Command button, the program reads the MCI command from the textbox and sends it to the MCI interface with the `mciSendString()` function. It then displays the function's return code, the string that corresponds to the return code, and the return string. If the MCI command executed successfully (the error code returned by the `mciSendString()` function is zero), the MCI command is added to the Command History box.

The Command History is maintained in a TextBox control, and as mentioned earlier, you can copy any command from it and paste it in the Enter MCI Command Here box. Moreover, you can edit the contents of the Command History box. You can change the order of the commands and delete commands, but you shouldn't edit their arguments. You can use the MCITest application to generate a sequence of MCI commands to carry out a task, determine the correct sequence of MCI commands and their arguments, and copy them into your application.

Code C.9: The Execute Command Button

```
Private Sub Command1_Click()

    errorCode = mciSendStringA(MCIcommand.Text, returnStr, 255, 0)
    MCIerror.Caption = errorCode
    returnCode = mciGetErrorStringA(errorCode, errorStr, 255)
    MCIcode.Caption = returnStr
    MCISTR.Caption = errorStr
    If errorCode = 0 Then History.Text = History.Text & _
        MCIcommand.Text & Chr$(13) & Chr$(10)

End Sub
```

The *MCIerror*, *MCIcode*, and *MCISTR* entries are the names of the Label controls in which the corresponding return codes and messages are displayed. In the Form's declaration section, the usual MCI functions are declared:

```
Private Declare Function mciSendStringA Lib "WinMM" (ByVal _
    MCIcommand As String, ByVal returnStr As String, ByVal _
    returnLength As Integer, ByVal callBack As Integer) As Long

Private Declare Function mciGetErrorStringA Lib "WinMM" (ByVal _
    error As Long, ByVal buffer As String, _
    ByVal length As Integer) As Integer
```

The `mciSendCommand()` Function

The MCI command message interface gives you lower-level access to multimedia devices than the command string interface does. It's also more difficult to use than the command string interface. With the command message interface, Windows doesn't need to break the command strings into command messages. By providing the messages directly, Windows can speed up the time required to process the commands.

To pass a message directly to an MCI device, use the `mciSendCommand()` function as follows:

```
Declare Function mciSendCommandA Lib "WinMM" (ByVal deviceID As _
    Integer, ByVal message As Integer, ByVal param1 As Long, _
    param2 As Any ) As Long
```

The *deviceID* argument is an integer that specifies the device ID, and *message* is another integer that specifies the command to be carried out. The *param1* argument is a Long value that contains the various arguments, and *param2* is a data structure that contains the information needed by the command. If the command requests information about the device, the information is reported back via the *param2* data structure.

Table C.5 lists the basic MCI command messages; they are the same as the MCI commands but have different names. See the Win32 Software Development Kit for the specific structure that needs to be passed with each command.

TABLE C.5: Basic MCI Command Messages

COMMAND	DESCRIPTION
MCI_CLOSE	Closes the device
MCI_OPEN	Initializes the device
MCI_PLAY	Starts transferring data to device
MCI_PAUSE	Pauses playing
MCI_STOP	Stops playing or recording
MCI_GETDEVCAPS	Obtains the capabilities of a device
MCI_INFO	Obtains information from a device
MCI_STATUS	Obtains status information from a device
MCI_LOAD	Loads data from a file

Continued on next page

TABLE C.5 CONTINUED: Basic MCI Command Messages

COMMAND	DESCRIPTION
MCI_RECORD	Starts recording
MCI_RESUME	Resumes playing on a paused device
MCI_SAVE	Saves data to a file
MCI_SEEK	Seeks forward or backward
MCI_SET	Sets the operating state of a device

VB6 at Work: The PlayWave Project

The PlayWave application demonstrates how to set up the data structures and call the `mciSendCommand()` function to play a WAV file. The application contains a Command button that calls the following `PlayWave()` subroutine:

```
Private Sub Sound_Click()
    PlayWave App.Path & "\howareu.wav"
End Sub
```

You may have to adjust the path name of the sound file.

The `PlayWave()` subroutine initializes the data structures needed for calling the `mciSendCommand()` function. Let's start with the declarations, constant definition, and data types.

Code C.10: The `mciSendCommand()` Function and Its Data Types

```
Private Declare Function mciSendCommandA Lib "WinMM" _
    (ByVal wDeviceID As Long, ByVal Message As Long, _
    ByVal dwParam1 As Long, dwParam2 As Any) As Long

Const MCI_OPEN = &H803
Const MCI_CLOSE = &H804
Const MCI_PLAY = &H806
Const MCI_OPEN_TYPE = &H2000&
Const MCI_OPEN_ELEMENT = &H200&
Const MCI_WAIT = &H2&

Private Type MCI_WAVE_OPEN_PARMS
    dwCallback As Long
    wDeviceID As Long
```

```

    lpstrDeviceType As String
    lpstrElementName As String
    lpstrAlias As String
    dwBufferSeconds As Long
End Type

Private Type MCI_PLAY_PARMS
    dwCallback As Long
    dwFrom As Long
    dwTo As Long
End Type

```

Finally, here's the code of the PlayWave() subroutine, which initializes the data structures and calls the mciSendCommand() function.

Code C.11: The PlayWave() Subroutine

```

Sub PlayWave(WaveFile As String)

    Dim errorCode As Integer
    Dim returnStr As String * 256
    Dim errorStr As String * 256
    Dim MCIWaveOpenParms As MCI_WAVE_OPEN_PARMS
    Dim MCIPlayParms As MCI_PLAY_PARMS

    MCIWaveOpenParms.dwCallback = 0
    MCIWaveOpenParms.wDeviceID = 0

    MCIWaveOpenParms.lpstrDeviceType = "waveaudio"
    MCIWaveOpenParms.lpstrElementName = WaveFile

    MCIWaveOpenParms.lpstrAlias = 0
    MCIWaveOpenParms.dwBufferSeconds = 0
    ' Open the device
    errorCode = mciSendCommandA(0, MCI_OPEN, MCI_OPEN_TYPE Or _
        MCI_OPEN_ELEMENT, MCIWaveOpenParms)
    If errorCode = 0 Then
        MCIPlayParms.dwCallback = 0
        MCIPlayParms.dwFrom = 0
        MCIPlayParms.dwTo = 0
        ' Play the wave file
        errorCode = mciSendCommandA(MCIWaveOpenParms.wDeviceID, MCI_PLAY, _
            MCI_WAIT, MCIPlayParms)
    End If
End Sub

```



```
' and close the device
    errorCode = mciSendCommandA(MCIWaveOpenParms.wDeviceID, _
                                MCI_CLOSE, 0, 0)
End If
End Sub
```

The `mciSendCommand()` function has four parameters as discussed above. The default waveaudio device's ID is 0. The `MCI_OPEN_TYPE` specifies a device type name that's defined in the `lpstrDeviceType` member of the `MCI_WAVE_OPEN_PARMS` structure. The `MCI_PLAY_PARMS` structure contains the information required for the file's playback. This structure only indicates the position to play from and the position to play to. The device ID, `wDeviceID`, in the `MCI_WAVE_OPEN_PARMS` structure is used for the first parameter. To close the device, you use the `MCI_CLOSE` command again with the same `wDeviceID`.

More API Sound Functions

The `mciSendString()` and `mciSendCommand()` API functions are the two most important functions for playing audio files. However, there are three more API functions for manipulating sounds: `MessageBeep()`, `sndPlaySound()`, and `PlaySound()`.

MessageBeep()

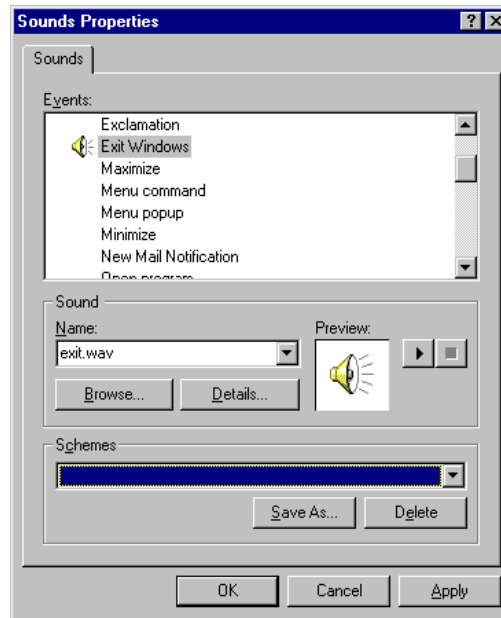
Use the `MessageBeep()` function to play a sound associated with one of Windows' alert events:

- Asterisk
- Critical stop
- Default beep
- Question
- Exclamation
- Windows exit
- Windows start

You can assign any sound stored in your system to these events through the Sounds program in the Control Panel. If you double-click the Sounds icon in the Control Panel, you'll see the window shown in Figure C.13.

FIGURE C.13:

The Sounds Properties window allows you to assign sounds to Windows alert events.



The syntax of the MessageBeep function is as follows:

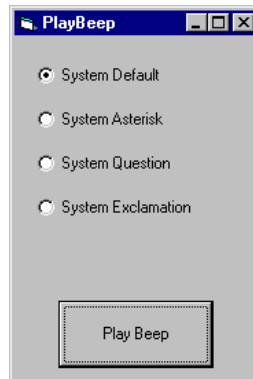
```
Private Declare Function MessageBeep Lib "User32" (ByVal alertLevel _  
    As Integer) As Integer
```

The *alertLevel* argument is an integer that identifies an alert level.

The PlayBeep application, shown in Figure C.14, shows you how to play a few of the sounds assigned to the Windows standard events.

FIGURE C.14:

The PlayBeep application



The events demonstrated in this example are identified by the following constants:

- *MB_OK*
- *MB_ICONQUESTION*
- *MB_ICONEXCLAMATION*
- *MB_ICONASTERISK*

You use one of these constants as the argument for `MessageBeep()`. The sample program shows you how to declare the function and set up the constants. The function returns a non-zero value if successful; otherwise, it returns zero.

Code C.12: The PlayBeep Application

```
Private Declare Function MessageBeep Lib "User32" (ByVal alertLevel _
    As Integer) As Integer

Const MB_OK = 0
Const MB_ICONQUESTION = &H30
Const MB_ICONASTERISK = &H40
Const MB_ICONEXCLAMATION = &H20

Private Sub Play_Click()

    If SndOption(0) = -1 Then MessageBeep (MB_OK)
    If SndOption(1) = -1 Then MessageBeep (MB_ICONASTERISK)
    If SndOption(2) = -1 Then MessageBeep (MB_ICONQUESTION)
    If SndOption(3) = -1 Then MessageBeep (MB_ICONEXCLAMATION)

End Sub
```

sndPlaySound()

The `sndPlaySound()` function can play any waveform file. Its syntax is as follows:

```
Function sndPlaySound(ByVal nameString As String, ByVal flags As _
    Integer) As Integer
```

This function takes two arguments:

- **nameString** The name of the WAV file to be played
- A list of options, which can be a combination of the flags listed in Table C.6.

TABLE C.6: Flags of the `sndPlaySound()` Function

FLAG	DESCRIPTION
SND_ASYNC	The sound plays asynchronously, which means the function returns immediately. If this parameter is not specified, the function starts the playback and returns control to the calling program. The sound continues playing.
SND_LOOP	The sound plays repeatedly. To stop the playback, call <code>sndPlaySound()</code> without a filename and with the <code>SND_ASYNC</code> and <code>SND_LOOP</code> parameters.
SND_SYNC	The sound plays synchronously, which means that the function returns control to the calling program only after the sound is finished playing
SND_MEMORY	Specifies that the waveform sound is in memory (it can be placed there with other API functions)
SND_NOSTOP	Indicates that the function won't restart the sound it is already playing
SND_NODEFAULT	If the sound is not found, the function returns and does not play the default Windows sound

To declare the function and the constants that correspond to its *flags* argument, include the following lines in your application:

Option Explicit

```
Private Declare Function sndPlaySoundA Lib "WinMM" _
    (ByVal _
    nameString As String, ByVal flags As Integer) As Integer
```

```
Const SND_ASYNC = &H1
Const SND_LOOP = &H8
Const SND_MEMORY = &H4
Const SND_NODEFAULT = &H2
Const SND_NOSTOP = &H10
Const SND_SYNC = &H0
```

To play a WAV file, call the function with the file's name and the appropriate flags combination:

```
Dim errorCode As Integer

errorCode = sndPlaySoundA("hello.wav", SND_SYNC Or SND_NODEFAULT)
```

PlaySound()

The `PlaySound()` function is similar to `sndPlaySound()`. In addition to the `SND_ASYNC`, `SND_SYNC`, and `SND_NODEFAULT` flags, this function provides the additional flags that are described in Table C.7.

TABLE C.7: Additional Flags of the PlaySound() Function

FLAG	DESCRIPTION
SND_ALIAS	The sound name is in the system registry
SND_RESOURCE	The sound name is a resource name
SND_FILENAME	The sound name is the filename of a WAV file
SND_NOWAIT	If the device is busy, the function should fail

To use the PlaySound() function, type the following declaration and define its flags as constants:

```
' Function declaration
Private Declare Function PlaySound Lib "WinMM" _ (ByVal _
    nameString As String, ByVal handle As Integer, _
    ByVal flags As Integer) As Integer

' Constant declaration
Const SND_ALIAS = &H10000
Const SND_FILENAME = &H20000
Const SND_RESOURCE = &H40004
Const SND_NOWAIT = &H2000
Const SND_SYNC = &H0
Const SND_NODEFAULT = &H2

errorCode = PlaySound("hello.wav", 0, SND_SYNC)

If errorCode <> 0 Then
    {process error}
End If
```

Adding Voice Messages to Applications

A great way to enhance all types of applications is to add audio messages. Windows provides the Sounds utility which you can use to associate sounds and system events. You can easily add voice and other audio messages to your applications. The messages can be simple greetings, dates, or number readouts.

The first example, the ReadNums application, is a number-reading utility that can be used as a proofing tool. For example, you can design a program that reads out long sequences of numbers such as the contents of a spreadsheet. The second example, the ReadDate application, is a utility that reads out the current date.

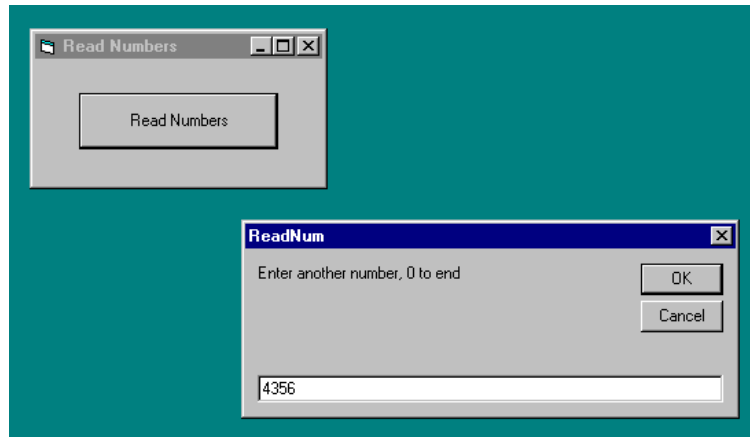
VB6 at Work: The ReadNums Project

In this example, we again use the `mciSendString()` function to playback individual words within a single WAV file. We use the *from* and *to* parameters of the play MCI command to play selected portions of the sound file. The example prompts the user to enter numbers with the `InputBox()` function and then reads out the number. You can easily modify the application to accept input from a grid, a text file, or any other source.

The ReadNums application (see Figure C.15) uses the Numbers.wav file, which contains the basic sounds needed to read out any number. The application can handle positive numbers up to 9,999, but you can easily modify the code to handle negative numbers, as well as numbers outside this range.

FIGURE C.15:

The ReadNums application reads out user-supplied numbers.



The application's complete code is given next. First, it reads the thousands as if they were single numbers and then the word *thousand*. It then reads the hundreds followed by the word *hundred*, and finally it reads the rest of the numbers that are smaller than 100.

Code C.13: The ReadNums Application

```
'This is the routine to read numbers
Private Sub ReadNum(number)

    Dim errorCode As Integer
    Dim returnStr As String * 256
    Dim returnCode As Integer
    Dim errorStr As String * 256
    Dim tenth As Integer
```

```

Dim leftover As Integer
Dim hundred As Integer
Dim thousand As Integer

If number < 20 Then      'Reads unique numbers
    ReadSingle (number)
ElseIf number < 100 Then 'Reads numbers less than 100
    tenth = number / 10
    ReadTenths (tenth * 10)
    leftover = number - (tenth * 10)
    If leftover > 0 Then
        ReadSingle (leftover)
    End If
ElseIf number < 1000 Then 'Reads numbers between 100 and 999

    hundred = number / 100
    ReadSingle (hundred)
    errorCode = mciSendStringA("play numbers.wav from 28000 to _
        29000 wait", returnStr, 255, 0)

    leftover = number - (hundred * 100)
    If leftover > 0 Then
        tenth = leftover / 10
        If tenth > 0 Then ReadTenths (tenth * 10)
        leftover = number - (hundred * 100) - (tenth * 10)
        If leftover > 0 Then
            ReadSingle (leftover)
        End If
    End If
Else          'Reads number between 1000 and 9999
    thousand = number / 1000
    ReadSingle (thousand)
    errorCode = mciSendStringA("play numbers.wav from 29000 to _
        30500 wait", returnStr, 255, 0)
    leftover = number - (thousand * 1000)

    If leftover > 0 Then
        hundred = leftover / 100
        If hundred > 0 Then
            ReadSingle (hundred)
            errorCode = mciSendStringA("play numbers.wav from 28000 to _
                29000 wait", returnStr, 255, 0)
        End If

        leftover = number - (thousand * 1000) - (hundred * 100)
        If leftover > 0 Then

```

```
tenth = leftover / 10
If tenth > 0 Then ReadTenths (tenth * 10)
leftover = number - (hundred * 100) - (tenth * 10)
If leftover > 0 Then
    ReadSingle (leftover)
End If
End If
End If
End If

End Sub
```

The ReadSingle() module reads numbers 1 through 19. The ReadTenths() procedure reads the numbers 20, 30, 40, 50, 60, 70, 80, and 90.

The ReadDate application, which you can find in the projects folder for this appendix, is quite similar. It uses the words in the files Days.wav, Months.wav, and Years.wav to read out dates. You can open the project in Visual Basic's editor and examine it. We will use the procedures of these applications in Chapter 21, *Visual Basic and the Web*, to build an ActiveX control that reads out numbers and dates.

Enhancing Applications with Video

To really extend your application to another level, you can add digitized video and animation. Digitizing video is fairly straightforward, but it requires a video capture board. Animation can be produced with the appropriate software. Whether you digitize video or produce animation with software, the result is usually an *AVI (Audio-Video Interleaved)* file, which can be played back with the ActiveX Movie control on any Windows 95/98 system.

Several types of digital video formats are on the market today. The native format for the Windows operating system is Microsoft's Video for Windows. The other formats include Apple Computer's QuickTime for Windows and MPEG.

Video can be stored in 8-bit, 16-bit, and 24-bit formats. As you might expect, storing video in 24-bit format is not practical since the image can be large. Video is also stored at a certain frame size; currently, the standard frame size is 320 × 240, or quarter-screen. The only limitations are the playback rate and available disk space.

We normally view movies and television at 30 frames a second, which is full-motion video. However, playback at 15 frames a second is sufficient for computer applications. Playback at higher rates requires special, expensive hardware.

Using Video for Windows

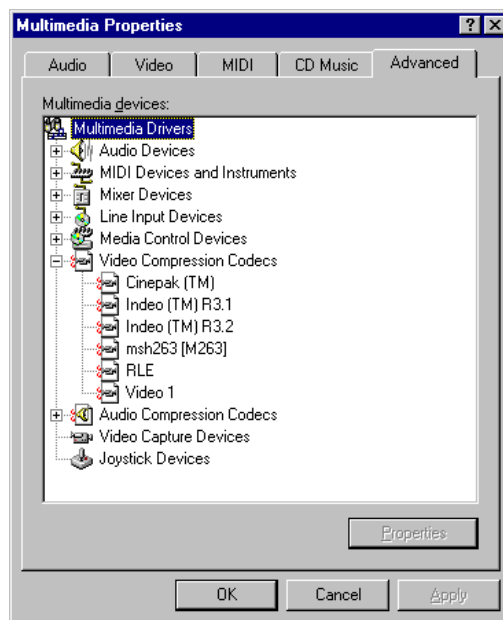
To solve the problems of playing video from Windows, Microsoft invented the Video for Windows format, which was introduced in Windows 3.1. The AVI format of Video for Windows means that audio and video information is interleaved on the track of the CD-ROM, which enables playback of synchronized sound and video. Video for Windows can playback AVI files without hardware support. Now Windows 95/98 ships with a 32-bit version that allows for the playback of 320×240 video.

To sustain an acceptable frame rate, Video for Windows must compress the digitized video a great deal. Windows comes with several video compression schemes. To see the available compression schemes, follow these steps:

1. Choose Start > Settings > Control Panel.
2. Double-click Multimedia, and then select the Advanced tab.
3. Double-click the Video Compression Codecs folder. Windows displays the Multimedia Properties window shown in Figure C.16. The video codecs include Cinepak, Indeo, Video 1, RLE and MSH. These are the codecs that can be used to encode and playback video files.

FIGURE C.16:

The video compression codecs for Windows



If you purchase a video capture card, you'll get VidCap and VidEdit, two utilities for capturing and editing video. The package allows you to play back, capture, and edit video.

Using MPEG

MPEG (Motion Picture Experts Group) is the most efficient codec for video compression and will probably emerge as the new standard for video compression. Until recently, MPEG-compressed video couldn't be played back without the aid of dedicated hardware. The speed of the Pentium™ processor and the Windows built-in support for MPEG software-only playback will lead to the replacement of AVI by the MPEG standard. MPEG playback cards are fairly inexpensive, but they are still required for anything more than stamp-size, jerky video playback. The problem with MPEG is that capturing video in this format requires expensive hardware, which isn't found on most PCs. When software-only MPEG playback becomes available, new possibilities for high-quality multimedia presentations will emerge, and MPEG recording prices will drop significantly.