```
      <r:RibbonApplicationMenuItem.Command>
        <r:RibbonCommand LabelTitle="Save As" Executed="SaveAs_Executed" />
       </r:RibbonApplicationMenuItem.Command>
    </r:RibbonApplicationMenuItem>

    <r:RibbonApplicationMenuItem>
      <r:RibbonApplicationMenuItem.Command>
        <r:RibbonCommand LabelTitle="Save" Executed="Save_Executed" />
       </r:RibbonApplicationMenuItem.Command>
    </r:RibbonApplicationMenuItem>
  </r:RibbonApplicationMenuItem>
```

■ **Note** Even when a submenu isn't being shown, the application menu keeps its second column. This area is reserved for a list of recent documents, like the one shown in the latest versions of Office and Windows 7 Paint. To create a recent document list of your own, you need to set the RibbonApplicationMenu.RecentItemList. If you don't, this area will simply remain blank.

Tabs, Groups, and Buttons

The ribbon uses the same model to fill its toolbar tabs as it does to fill its application menu, just with a few extra layers.

First, the ribbon holds a collection of tabs. In turn, each tab holds one or more groups, which is an outlined, titled, boxlike section of the ribbon. Lastly, each group holds one or more ribbon controls. Figure 25-11 shows this arrangement.

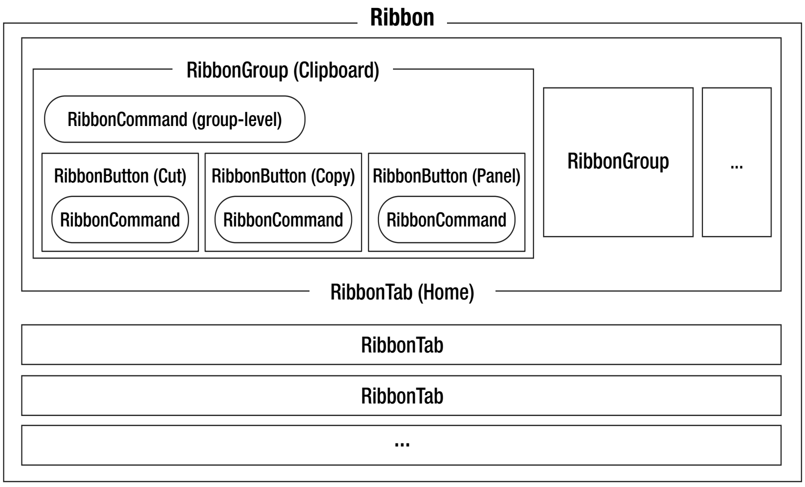

Figure 25-11. *Tabs, groups, and buttons*

Each of these ingredients has a corresponding class. To create a ribbon like the one shown in Figure 25-11, you start by declaring the appropriate RibbonTab objects, fill each one with RibbonGroup objects, and place ribbon controls (like the straightforward RibbonButton) in each group. As with the application menu, you match each control with a RibbonCommand object that defines its text and image content, and with the event handlers that handle clicks and determine command state.

In addition, you must attach a RibbonCommand object to each group. This RibbonCommand serves a few special purposes. First, the RibbonCommand.LabelTitle property sets the title of the group, which appears just under the group section on the ribbon. Second, the RibbonCommand.SmallImageSource property sets the image that will be used if space is limited and the group is collapsed down to a single button, as shown in Figure 25-12. Finally, the RibbonCommand.Executed event allows you to create a dialog launcher. (A *dialog launcher* is a tiny icon that appears at the bottom-right corner of some groups that, when clicked, pops up a dialog window with more options.) To add a dialog launcher to a group, set the RibbonGroup.HasDialogLauncher property to true, and handle the dialog launcher click by responding to the RibbonCommand.Executed event.

Here's a portion of ribbon markup that defines the Clipboard group and places three commands inside:

```
<r:Ribbon Title="Ribbon Test">
  <r:RibbonTab Label="Home">
    <r:RibbonTab.Groups>

      <r:RibbonGroup>
        <r:RibbonGroup.Command>
          <r:RibbonCommand LabelTitle="Clipboard"
           SmallImageSource="images/clipboard.png" />
        </r:RibbonGroup.Command>

        <r:RibbonButton Command="{StaticResource CutCommand}" />
        <r:RibbonButton Command="{StaticResource CopyCommand}" />
        <r:RibbonButton Command="{StaticResource PasteCommand}" />
      </r:RibbonGroup>
    </r:RibbonTab.Groups>
  </r:RibbonTab>
</r:Ribbon>
```

You'll notice that in this example the RibbonCommand objects aren't declared inline. This is the recommended approach, because it frees you up to use the same commands for multiple buttons of the ribbon, as well as in the application menu, the quick access toolbar, and directly in your application. Instead, the RibbonCommand objects are stored in the resources section of the window:

```
<r:RibbonWindow.Resources>
  <ResourceDictionary>
    <r:RibbonCommand x:Key="CutCommand" LabelTitle="Cut" ToolTipTitle="Cut"
     ToolTipDescription="Copies the selected text to the clipboard and removes it"
     SmallImageSource="images\cut.png" LargeImageSource="images\cut.png"
     CanExecute="CutCommand_CanExecute" Executed="CutCommand_Executed" />
    <r:RibbonCommand x:Key="CopyCommand" ... />
    <r:RibbonCommand x:Key="PasteCommand" ... />
    ...
  </ResourceDictionary>
</r:RibbonWindow.Resources>
```

■ **Note** An alternate but equally valid approach is to use a custom command class and connect the commands to the controls using command bindings. This slightly more elaborate technique, which may suit larger projects, is demonstrated in Chapter 9.

In this example, the ribbon was entirely made up of RibbonButton objects, which is the most common ribbon control type. However, WPF gives you several more options, which are outlined in Table 25-2. As with the application menu, most of the ribbon classes derive from the standard WPF controls. They simply implement the IRibbonControl interface to get more capabilities.

Table 25-2. *Ribbon Control Classes*

Name	Description
RibbonButton	A clickable text-and-image button, which is the most common ingredient on the ribbon.
RibbonCheckBox	A check box that can be checked or unchecked.
RibbonToggleButton	A button that has two states: pressed or unpressed. For example, many programs use this sort of button to turn on or off font characteristics such as bold, italic, and underline.
RibbonDropDownButton	A button that pops open a menu. You fill the menu with MenuItem objects using the RibbonDropDownButton.Items collection.
RibbonSplitButton	Similar to a RibbonDropDownButton, but the button is actually divided into two sections. The user can click the top portion (with the picture) to run the command or the bottom portion (with the text and drop-down arrow) to show the linked menu of items. For example, the Paste command in Word is a RibbonSplitButton.
RibbonComboBox	Embeds a combo box in the ribbon, which the user can use to type in text or make a selection, just as with the standard ComboBox control.
RibbonTextBox	Embeds a text box in the ribbon, which the user can use to type in text, just as with the standard TextBox control.
RibbonLabel	Embeds static text in the ribbon, just like the standard Label control. This is primarily useful when using embedded controls such as RibbonComboBox and RibbonTextBox, when you need a way to add descriptive captions.
RibbonSeparator	Draws a vertical line between individual controls (or groups of controls) in the ribbon.

Ribbon Sizing

One of the ribbon's most remarkable features is its ability to resize itself to fit the width of the window by reducing and rearranging the buttons in each groups.

When you create a ribbon with WPF, you get basic resizing for free. This resizing, which is built into the RibbonWrapPanel, uses a different template depending on the number of controls in the group and the size of the group. For example, a group with three RibbonButton objects will display them from left to right, if space permits. If not, the controls on the right are collapsed to small icons, then their text is stripped away to reclaim more space, and finally the whole group is reduced to a single button that, when clicked, shows all the commands in a drop-down list. Figure 25-12 illustrates this process with a ribbon that has three copies of the File group. The first is fully expanded, the second is partially collapsed, and the second is completely collapsed. (It's worth noting that in order to create this example, the ribbon must be explicitly configured to not collapse the first group. Otherwise, it will always try to partially collapse every group before it fully collapses any group.)

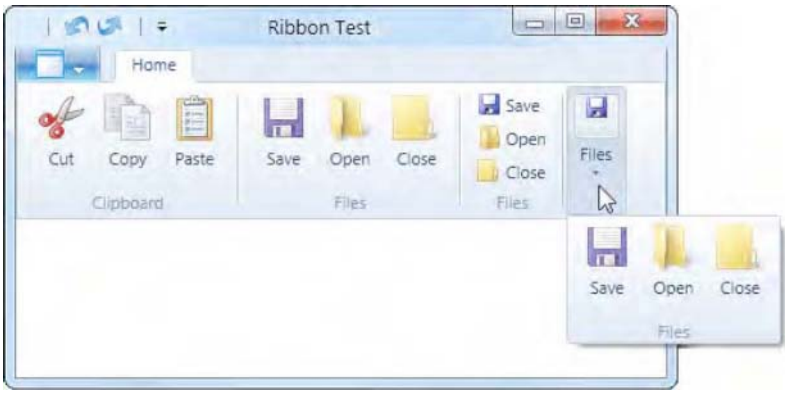

Figure 25-12. *Shrinking the ribbon*

You can use several techniques to change the sizing of a ribbon group. You can use the RibbonTab.GroupSizeReductionOrder property to set which groups should be reduced first. You specify each group using the value of its LabelTitle. Here's an example:

```
<r:RibbonTab Label="Home" GroupSizeReductionOrder="Clipboard,Tasks,File">
```

As you reduce the size of the window, all the groups will be collapsed bit by bit. However, the Clipboard group will switch to a more compact layout first, followed by the Tasks group, and so on. If you continue shrinking the window, there will be another round of group rearrangement, once again led by the Clipboard group. If you don't set the GroupSizeReductionOrder property, the rightmost group leads the way.

A more powerful approach is to create a collection of RibbonGroupSizeDefinition objects that dictates how a group should collapse itself. Each RibbonGroupSizeDefinition is a template that defines a single layout. It specifies which commands should get large icons, which ones should get small icons, and which ones should include display text. Here's an example of a RibbonControlSizeDefinition that sets the layout for a group of four controls, making them all as big as can be:

```
<r:RibbonGroupSizeDefinition>
  <r:RibbonControlSizeDefinition ImageSize="Large" IsLabelVisible="True" />
  <r:RibbonControlSizeDefinition ImageSize="Large" IsLabelVisible="True" />
  <r:RibbonControlSizeDefinition ImageSize="Large" IsLabelVisible="True" />
  <r:RibbonControlSizeDefinition ImageSize="Large" IsLabelVisible="True" />
</r:RibbonGroupSizeDefinition>
```

To take control of group resizing, you need to define multiple RibbonGroupSizeDefinition objects and order them from largest to smallest in a RibbonGroupSizeDefinitionCollection. As the group is collapsed, the ribbon can then switch from one layout to the next to reclaim more space, while keeping the layout you want (and ensuring that the controls you think are most important remain visible). Usually, you'll place the RibbonGroupSizeDefinitionCollection in the Ribbon.Resources section, so you can reuse the same sequences of templates for more than one four-button group.

```
<r:Ribbon.Resources>
  <r:RibbonGroupSizeDefinitionCollection x:Key="RibbonLayout">

    <!-- All large controls. -->
    <r:RibbonGroupSizeDefinition>
      <r:RibbonControlSizeDefinition ImageSize="Large" IsLabelVisible="True"/>
      <r:RibbonControlSizeDefinition ImageSize="Large" IsLabelVisible="True"/>
      <r:RibbonControlSizeDefinition ImageSize="Large" IsLabelVisible="True"/>
      <r:RibbonControlSizeDefinition ImageSize="Large" IsLabelVisible="True"/>
    </r:RibbonGroupSizeDefinition>

    <!-- A large control at both ends, with two small controls in between. -->
    <r:RibbonGroupSizeDefinition>
      <r:RibbonControlSizeDefinition ImageSize="Large" IsLabelVisible="True"/>
      <r:RibbonControlSizeDefinition ImageSize="Small" IsLabelVisible="True"/>
      <r:RibbonControlSizeDefinition ImageSize="Small" IsLabelVisible="True"/>
      <r:RibbonControlSizeDefinition ImageSize="Large" IsLabelVisible="True"/>
    </r:RibbonGroupSizeDefinition>

    <!-- Same as before, but now with no text for the small buttons. -->
    <r:RibbonGroupSizeDefinition>
      <r:RibbonControlSizeDefinition ImageSize="Large" IsLabelVisible="True"/>
      <r:RibbonControlSizeDefinition ImageSize="Small" IsLabelVisible="False"/>
      <r:RibbonControlSizeDefinition ImageSize="Small" IsLabelVisible="False"/>
      <r:RibbonControlSizeDefinition ImageSize="Large" IsLabelVisible="True"/>
    </r:RibbonGroupSizeDefinition>

    <!-- All small buttons. -->
    <r:RibbonGroupSizeDefinition>
      <r:RibbonControlSizeDefinition ImageSize="Small" IsLabelVisible="True"/>
      <r:RibbonControlSizeDefinition ImageSize="Small" IsLabelVisible="False"/>
      <r:RibbonControlSizeDefinition ImageSize="Small" IsLabelVisible="False"/>
      <r:RibbonControlSizeDefinition ImageSize="Small" IsLabelVisible="True"/>
    </r:RibbonGroupSizeDefinition>

    <!-- All small, no-text buttons. -->
    <r:RibbonGroupSizeDefinition>
      <r:RibbonControlSizeDefinition ImageSize="Small" IsLabelVisible="False"/>
      <r:RibbonControlSizeDefinition ImageSize="Small" IsLabelVisible="False"/>
      <r:RibbonControlSizeDefinition ImageSize="Small" IsLabelVisible="False"/>
      <r:RibbonControlSizeDefinition ImageSize="Small" IsLabelVisible="False"/>
    </r:RibbonGroupSizeDefinition>
    <!-- Collapse the entire group to a single drop-down button. -->
    <r:RibbonGroupSizeDefinition IsCollapsed="True" />
  </r:RibbonGroupSizeDefinitionCollection>
</r:Ribbon.Resources>
```

The Quick access Toolbar

The final ingredient that you'll consider in the ribbon is the quick access toolbar (or QAT). It's a narrow strip of commonly used buttons that sits either just above or just below the rest of the ribbon, depending on user selection.

The QAT is represented by the QuickAccessToolBar object, which holds a series of RibbonButton objects. When defining the RibbonCommand for these objects, you need only supply the tooltip text and small image, because text labels and large images are never shown.

The only new detail in the QAT is the customize menu that appears when you click the drop-down arrow off the far right of it (Figure 25-13). You can use this menu to let users customize the commands that appear in the QAT. Or, you can disable the customize menu by setting QuickAccessToolBar.CanUserCustomize to false.

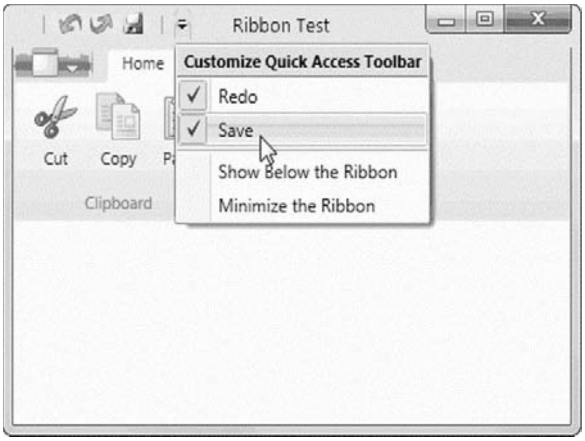

Figure 25-13. *The quick access toolbar*

The user customization works through the attached RibbonQuickAccessToolBar.Placement property. You have three choices. Use InToolBar if you want the command to appear in the QAT only (not in the customize menu) so that it's always visible. Use InCustomizeMenuAndToolBar if you want the command to appear in the QAT and the customize menu so the user has the ability to uncheck the command and hide it. Use InCustomizeMenu if you want the command to appear, unchecked, in the customize menu but not in the ribbon, which means the user can explicitly choose to show it if needed.

Here's the definition for a simple QAT:

```
<r:Ribbon.QuickAccessToolBar>
  <r:RibbonQuickAccessToolBar CanUserCustomize="True">

    <!-- Always visible and can't be removed. -->
    <r:RibbonButton Command="{StaticResource UndoCommand}"
     r:RibbonQuickAccessToolBar.Placement="InToolBar" />

    <!-- Visible, but can be hidden using the customize menu. -->
    <r:RibbonButton Command="{StaticResource RedoCommand}"
     r:RibbonQuickAccessToolBar.Placement="InCustomizeMenuAndToolBar" />

    <!-- Not visible, but can be shown using the customize menu. -->
    <r:RibbonButton Command="{StaticResource SaveCommand}"
     r:RibbonQuickAccessToolBar.Placement="InCustomizeMenu" />

  </r:RibbonQuickAccessToolBar>
</r:Ribbon.QuickAccessToolBar>
```

The Last Word

In this chapter you looked at four controls that underpin professional Windows applications. The first three—the Menu, ToolBar, and StatusBar—derive from the ItemsControl class you considered in Chapter 20. But rather than display data, they hold groups of menu commands, toolbar buttons, and status items. This is one more example that shows how the WPF library takes fundamental concepts, such as the ItemsControl, and uses them to standardize entire branches of the control family.

The fourth and final control that you considered is the Ribbon, a toolbar replacement that was introduced as the distinguishing feature of Office 2007 and became a standard ingredient Windows 7. Although the ribbon isn't yet baked into the .NET runtime, its availability as a free library is an impressive win for WPF developers. It's far better than the situation that developers were in with earlier Microsoft user interface technologies such as Windows Forms, which were embarrassingly slow to adopt cutting-edge features from Office and other Windows applications.

CHAPTER 26

Sound and Video

In this chapter, you'll tackle two more areas of WPF functionality: audio and video.

The support WPF provides for audio is a significant step up from previous versions of .NET, but it's far from groundbreaking. WPF gives you the ability to play a wide variety of sound formats, including MP3 files and anything else supported by Windows Media Player. However, WPF's sound capabilities still fall far short of DirectSound (the advanced audio API in DirectX), which allows you to apply dynamic effects and place sounds in a simulated 3-D space. WPF also lacks a way to retrieve spectrum data that tells you the highs and lows of sound, which is useful for creating some types of synchronized effects and sound-driven animations.

WPF's video support is more impressive. Although the ability to play video (such as MPEG and WMV files) isn't earth-shattering, the way it integrates into the rest of the WPF model is dramatic. For example, you can use video to fill thousands of elements at once and combine it with effects, animation, transparency, and even 3-D objects.

In this chapter, you'll see how to integrate video and audio content into your applications. You'll even take a quick look at WPF's support for speech synthesis and speech recognition. But before you get to the more exotic examples, you'll begin by considering the basic code required to play humble WAV audio.

Playing WAV Audio

The .NET Framework has a sketchy history of sound support. Versions 1.0 and 1.1 didn't include any managed way to play audio, and when the long-delayed support finally appeared in .NET 2.0, it was in the form of the rather underwhelming SoundPlayer class (which you can find in the underpopulated System.Media namespace). The SoundPlayer is severely limited: it can play only WAV audio files, it doesn't support playing more than one sound at once, and it doesn't provide the ability to control any aspect of the audio playback (for example, details such as volume and balance). To get these features, developers using the Windows Forms toolkit had to work with the unmanaged quartz.dll library.

Note The quartz.dll library is a key part of DirectX, and it's included with Windows Media Player and the Windows operating system. (Sometimes, the same component is known by the more marketing-friendly term DirectShow, and previous versions were called ActiveMovie.) For the gory details that describe how to use quartz.dll with Windows Forms, refer to my book *Pro .NET 2.0 Windows Forms and Custom Controls in C#* (Apress, 2005).

The SoundPlayer class is supported in WPF applications. If you can live with its significant limitations, it still presents the easiest, most lightweight way to add audio to an application. The SoundPlayer class is also wrapped by the SoundPlayerAction class, which allows you to play sounds through a declarative trigger (rather than writing a few lines of C# code in an event handler). In the following sections, you'll take a quick look at both classes, before you move on to WPF's much more powerful MediaPlayer and MediaElement classes.

The SoundPlayer

To play a sound with the SoundPlayer class, you follow several steps:

1. Create a SoundPlayer instance.

2. Specify the sound content by setting either the SoundLocation property or the Stream property. If you have a file path that points to a WAV file, use the SoundLocation property. If you have a Stream-based object that contains WAV audio content, use the Stream property.

■ **Note** If your audio content is stored in a binary resource and embedded in your application, you'll need to access it as a stream (see Chapter 7) and use the SoundPlayer.Stream property. That's because the SoundPlayer doesn't support WPF's pack URI syntax.

---

3. Once you've set the Stream or SoundLocation property, you can tell SoundPlayer to actually load the audio data by calling the Load() or LoadAsync() method. The Load() method is the simplest—it stalls your code until all the audio is loaded into memory. LoadAsync() quietly carries its work out on another thread and fires the LoadCompleted event when it's finished.

■ **Note** Technically, you don't need to use Load() or LoadAsync(). The SoundPlayer will load the audio data if needed when you call Play() or PlaySync(). However, it's a good idea to explicitly load the audio—not only does that save you the overhead if you need to play it multiple times, but it also makes it easy to handle exceptions related to file problems separately from exceptions related to audio playback problems.

---

4. Now, you can call PlaySync() to pause your code while the audio plays, or you can use Play() to play the audio on another thread, ensuring that your application's interface remains responsive. Your only other option is PlayLooping(), which plays the audio asynchronously in an unending loop (perfect for those annoying soundtracks). To halt the current playback at any time, just call Stop().

The following code snippet shows the simplest approach to load and play a sound asynchronously:

```
SoundPlayer player = new SoundPlayer();
player.SoundLocation = "test.wav";
try
{
    player.Load();
    player.Play();
}
catch (System.IO.FileNotFoundException err)
{
    // An error will occur here if the file can't be found.
}
catch (FormatException err)
{
    // A FormatException will occur here if the file doesn't
    // contain valid WAV audio.
}
```

So far, the code has assumed that the audio is present in the same directory as the compiled application. However, you don't need to load the SoundPlayer audio from a file. If you've created small sounds that are played at several points in your application, it may make more sense to embed the sound files into your compiled assembly as a binary resource (not to be confused with declarative resources, which are the resources you define in XAML markup). This technique, which was discussed in Chapter 11, works just as well with sound files as it does with images. For example, if you add the ding.wav audio file with the resource name Ding (just browse to the Properties → Resources node in the Solution Explorer and use the designer support), you could use this code to play it:

```
SoundPlayer player = new SoundPlayer();
player.Stream = Properties.Resources.Ding;
player.Play();
```

■ **Note** The SoundPlayer class doesn't deal well with large audio files, because it needs to load the entire file into memory at once. You might think that you can resolve this problem by submitting a large audio file in smaller chunks, but the SoundPlayer wasn't designed with this technique in mind. There's no easy way to synchronize the SoundPlayer so that it plays multiple audio snippets one after the other, because it doesn't provide any sort of queuing feature. Each time you call PlaySync() or Play(), the current audio playback stops. Workarounds are possible, but you'll be far better off using the MediaElement class discussed later in this chapter.

The SoundPlayerAction

The SoundPlayerAction makes it more convenient to use the SoundPlayer class. The SoundPlayerAction class derives from TriggerAction (Chapter 11), which allows you to use it in response to any event.

Here's a button that uses a SoundPlayerAction to connect the Click event to a sound. The trigger is wrapped in a style that you could apply to multiple buttons (if you pulled it out of the button and placed it in a Resources collection).

```
<Button>
  <Button.Content>Play Sound</Button.Content>
  <Button.Style>
    <Style>
      <Style.Triggers>
        <EventTrigger RoutedEvent="Button.Click">
          <EventTrigger.Actions>
            <SoundPlayerAction Source="test.wav"></SoundPlayerAction>
          </EventTrigger.Actions>
        </EventTrigger>
      </Style.Triggers>
    </Style>
  </Button.Style>
</Button>
```

When using the SoundPlayerAction, the sound is always played asynchronously.

System Sounds

One of the shameless frills of the Windows operating system is its ability to map audio files to specific system events. Along with SoundPlayer, WPF also includes a System.Media.SystemSounds class that allows you to access the most common of these sounds and use them in your own applications. This technique works best if all you want is a simple chime to indicate the end of a long-running operation or an alert sound to indicate a warning condition.

Unfortunately, the SystemSounds class is based on the MessageBeep Win32 API, and as a result, it provides access only to the following generic system sounds:

- Asterisk
- Beep
- Exclamation
- Hand
- Question

The SystemSounds class provides a property for each of these sounds, which returns a SystemSound object you can use to play the sound through its Play() method. For example, to sound a beep in your code, you simply need to execute this line of code:

```
SystemSounds.Beep.Play();
```

To configure what WAV files are used for each sound, head to the Control Panel, and select the Sounds and Audio Devices icon (in Windows XP) or the Sound icon (in Windows Vista or Windows 7).

The MediaPlayer

The SoundPlayer, SoundPlayerAction, and SystemSounds classes are easy to use but relatively underpowered. In today's world, it's much more common to use compressed MP3 audio for everything except the simplest of sounds, instead of the original WAV format. But if you want to play MP3 audio or MPEG video, you need to turn to two different classes: MediaPlayer and MediaElement. Both classes depend on key pieces of technology that are provided through Windows Media Player.

The MediaPlayer class (found in the WPF-specific System.Windows.Media namespace) is the WPF equivalent to the SoundPlayer class. Although it's clearly not as lightweight, it works in a similar way—

namely, you create a MediaPlayer object, call the Open() method to load your audio file, and call Play() to begin playing it asynchronously. (There's no option for synchronous playback.) Here's a bare-bones example:

```
private MediaPlayer player = new MediaPlayer();

private void cmdPlayWithMediaPlayer_Click(object sender, RoutedEventArgs e)
{
    player.Open(new Uri("test.mp3", UriKind.Relative));
    player.Play();
}
```

There are a few important details to notice in this example:

- The MediaPlayer is created outside the event handler, so it lives for the lifetime of the window. That's because the MediaPlayer.Close() method is called when the MediaPlayer object is disposed from memory. If you create a MediaPlayer object in the event handler, it will be released from memory almost immediately and probably garbage collected shortly after, at which point the Close() method will be called and playback will be halted.

■ **Tip** You should create a Window.Unloaded event handler to call Close() to stop any currently playing audio when the window is closed.

- You supply the location of your file as a URI. Unfortunately, this URI doesn't use the application pack syntax that you learned about in Chapter 7, so it's not possible to embed an audio file and play it using the MediaPlayer class. This limitation is because the MediaPlayer class is built on functionality that's not native to WPF—instead, it's provided by a distinct, unmanaged component of the Windows Media Player.
- There's no exception handling code. Irritatingly, the Open() and Play() methods don't throw exceptions (the asynchronous load and playback process is partly to blame). Instead, it's up to you to handle the MediaOpened and MediaFailed events if you want to determine whether your audio is being played.

The MediaPlayer is fairly straightforward but still more capable than SoundPlayer. It provides a small set of useful methods, properties, and events. Table 26-1 has the full list.

Table 26-1. *Key MediaPlayer Members*

Member	Description
Balance	Sets the balance between the left and right speaker as a number from –1 (left speaker only) to 1 (right speaker only).

Member	Description
Volume	Sets the volume as a number from 0 (completely muted) to 1 (full volume). The default value is 0.5.
SpeedRatio	Sets a speed multiplier to play audio (or video) at faster than normal speed. The default value of 1 is normal speed, while 2 is two-times normal speed, 10 is ten-times speed, 0.5 is half-times speed, and so on. You can use any positive double value.
HasAudio and HasVideo	Indicates whether the currently loaded media file includes audio or video, respectively. To show video, you need to use the MediaElement class described in the next section.
NaturalDuration, NaturalVideoHeight, and NaturalVideoWidth	Indicates the play duration at normal speed and the size of the video window. (As you'll discover later, you can scale or stretch a video to fit different window sizes.)
Position	A TimeSpan indicating the current location in the media file. You can set this property to skip to a specific time position.
DownloadProgress and BufferingProgress	Indicates the percentage of a file that has been downloaded (useful if theSource is a URL pointing to a web or remote computer) or buffered (if the media file you're using is encoded in a streaming format so it can be played before it's entirely downloaded). The percentage is represented as a number from 0 to 1.
Clock	Gets or sets the MediaClock that's associated with this player. The MediaClock is used only when you're synchronizing audio to a timeline (in much the same way that you learned to synchronize an animation to a timeline in Chapter 15). If you're using the methods of the MediaPlayer to perform manual playback, this property is null.
Open()	Loads a new media file.
Play()	Begins playback. Has no effect if the file is already being played.
Pause()	Pauses playback but doesn't change the position. If you call Play() again, playback will begin at the current position. Has no effect if the audio is not playing.
Stop()	Stops playback and resets the position to the beginning of the file. If you call Play() again, playback will begin at the beginning of the file. Has no effect if the audio has already been stopped.

Using these members, you could build a basic but full-featured media player. However, WPF programmers usually use another quite similar element, which is defined in the next section: the MediaElement class.

The MediaElement

The MediaElement is a WPF element that wraps all the functionality of the MediaPlayer class. Like all elements, the MediaElement is placed directly in your user interface. If you're using the MediaElement to play audio, this fact isn't important, but if you're using the MediaElement for video, you place it where the video window should appear.

A simple MediaElement tag is all you need to play a sound. For example, if you add this markup to your user interface:

```
<MediaElement Source="test.mp3"></MediaElement>
```

the test.mp3 audio will be played as soon as it's loaded (which is more or less as soon as the window is loaded).

Playing Audio Programmatically

Usually, you'll want the ability to control playback more precisely. For example, you might want it to be triggered at a specific time, repeated indefinitely, and so on. One way to achieve this result is to use the methods of the MediaElement class at the appropriate time.

The startup behavior of the MediaElement is determined by its LoadedBehavior property, which is one of the few properties that the MediaElement class adds, which isn't found in the MediaPlayer class. The LoadedBehavior takes any value from the MediaState enumeration. The default value is Play, but you can also use Manual, in which case the audio file is loaded, and your code takes responsibility for starting the playback at the right time. Another option is Pause, which also suspends playback but doesn't allow you to use the playback methods. (Instead, you'll need to start playback using triggers and a storyboard, as described in the next section.)

■ **Note** The MediaElement class also provides an UnloadedBehavior property, which determines what should happen when the element is unloaded. In this case, Close is really the only sensible choice, because it closes the file and releases all system resources.

So to play audio programmatically, you must begin by changing the LoadedBehavior, as shown here:

```
<MediaElement Source="test.mp3" LoadedBehavior="Manual" Name="media"></MediaElement>
```

You must also choose a name so that you can interact with the media element in code. Generally, interaction consists of the straightforward Play(), Pause(), and Stop() methods. You can also set Position to move through the audio. Here's a simple event handler that seeks to the beginning and starts playback:

```
private void cmdPlay_Click(object sender, RoutedEventArgs e)
{
    media.Position = TimeSpan.Zero;
    media.Play();
}
```

If this code runs while playback is already underway, the first line will reset the position to the beginning, and playback will continue from that point. The second line will have no effect, because the media file is already being played. If you try to use this code on a MediaElement that doesn't have the LoadedBehavior property set to Manual, you'll receive an exception.

■ **Note** In a typical media player, you can trigger basic commands like play, pause, and stop in more than one way. Obviously, this is a great place to use the WPF command model. In fact, there's a command class that already includes some handy infrastructure, the System.Windows.Input.MediaCommands class. However, the MediaElement does not have any default command bindings that support the MediaCommands class. In other words, it's up to you to write the event handling logic that implements each command and calls the appropriate MediaElement method. The savings to you is that multiple user interface elements can be hooked up to the same command, reducing code duplication. Chapter 9 has more about commands.

Handling Errors

The MediaElement doesn't throw an exception if it can't find or load a file. Instead, it's up to you to handle the MediaFailed event. Fortunately, this task is easy. Just tweak your MediaElement tag:

```
<MediaElement ... MediaFailed="media_MediaFailed"></MediaElement>
```

And, in the event handler, use the ExceptionRoutedEventArgs.ErrorException property to get an exception object that describes the problem:

```
private void media_MediaFailed(object sender, ExceptionRoutedEventArgs e)
{
    lblErrorText.Content = e.ErrorException.Message;
}
```

Playing Audio with Triggers

So far, you haven't received any advantage by switching from the MediaPlayer to the MediaElement class (other than support for video, which is discussed later in this chapter). However, by using a MediaElement, you also gain the ability to control audio declaratively, through XAML markup rather than code. You do this using triggers and storyboards, which you first saw when you considered animation in Chapter 15. The only new ingredient is the MediaTimeline, which controls the timing of your audio or video file and works with MediaElement to coordinate its playback. MediaTimeline derives from Timeline and adds a Source property that identifies the audio file you want to play.

The following markup demonstrates a simple example. It uses the BeginStoryboard action to begin playing a sound when the mouse clicks a button. (Obviously, you could respond equally well to other mouse and keyboard events.)

```
<Grid>
 <Grid.RowDefinitions>
```

```
   <RowDefinition Size="Auto"></RowDefinition>
   <RowDefinition Size="Auto"></RowDefinition>
  </Grid.RowDefinitions>
  <MediaElement x:Name="media"></MediaElement>

  <Button>
    <Button.Content>Click me to hear a sound.</Button.Content>
    <Button.Triggers>
      <EventTrigger RoutedEvent="Button.Click">
        <EventTrigger.Actions>
        <BeginStoryboard>
          <Storyboard>
            <MediaTimeline Source="soundA.wav"
             Storyboard.TargetName="media"></MediaTimeline>
          </Storyboard>
        </BeginStoryboard>
        </EventTrigger.Actions>
      </EventTrigger>
    </Button.Triggers>
  </Button>
</Grid>
```

Because this example plays audio, the positioning of the MediaElement isn't important. In this example, it's placed inside a Grid, behind a Button. (The ordering isn't important, because the MediaElement won't have any visual appearance at runtime.) When the button is clicked, a Storyboard is created with a MediaTimeline. Notice that the source isn't specified in the MediaElement.Source property. Instead, the source is passed along through the MediaTimeline.Source property.

■ **Note** When you use MediaElement as the target of a MediaTimeline, it no longer matters what you set the LoadedBehavior and UnloadedBehavior to. Once you use a MediaTime, your audio or video is driven by a WPF animation clock (technically, an instance of the MediaClock class, which is exposed through the MediaElement.Clock property).

You can use a single Storyboard to control the playback of a single MediaElement—in other words, not only stopping it but also pausing, resuming, and stopping it at will. For example, consider the extremely simple four-button media player shown in Figure 26-1.

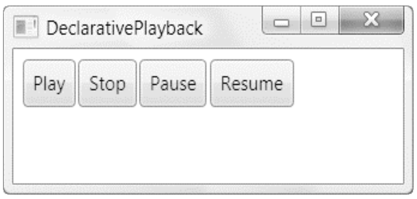

Figure 26-1. *A window for controlling playback*

This window uses a single MediaElement, MediaTimeline, and Storyboard. The Storyboard and MediaTimeline are declared in the Window.Resources collection:

```
<Window.Resources>
  <Storyboard x:Key="MediaStoryboardResource">
    <MediaTimeline Storyboard.TargetName="media" Source="test.mp3"></MediaTimeline>
    </Storyboard>
</Window.Resources>
```

The only challenge is that you must remember to define all the triggers for managing the storyboard in one collection. You can then attach them to the appropriate controls using the EventTrigger.SourceName property.

In this example, the triggers are all declared inside the StackPanel that holds the buttons. Here are the triggers and the buttons that use them to manage the audio:

```
<StackPanel Orientation="Horizontal">
  <StackPanel.Triggers>
    <EventTrigger RoutedEvent="ButtonBase.Click" SourceName="cmdPlay">
      <EventTrigger.Actions>
        <BeginStoryboard Name="MediaStoryboard"
         Storyboard="{StaticResource MediaStoryboardResource}"/>
      </EventTrigger.Actions>
    </EventTrigger>
    <EventTrigger RoutedEvent="ButtonBase.Click" SourceName="cmdStop">
      <EventTrigger.Actions>
        <StopStoryboard BeginStoryboardName="MediaStoryboard"/>
      </EventTrigger.Actions>
    </EventTrigger>
    <EventTrigger RoutedEvent="ButtonBase.Click" SourceName="cmdPause">
      <EventTrigger.Actions>
        <PauseStoryboard BeginStoryboardName="MediaStoryboard"/>
      </EventTrigger.Actions>
    </EventTrigger>
    <EventTrigger RoutedEvent="ButtonBase.Click" SourceName="cmdResume">
      <EventTrigger.Actions>
        <ResumeStoryboard BeginStoryboardName="MediaStoryboard"/>
      </EventTrigger.Actions>
    </EventTrigger>
  </StackPanel.Triggers>

  <MediaElement  Name="media"></MediaElement>
  <Button Name="cmdPlay">Play</Button>
  <Button Name="cmdStop">Stop</Button>
  <Button Name="cmdPause">Pause</Button>
  <Button Name="cmdResume">Resume</Button>
</StackPanel>
```

Notice that even though the implementation of MediaElement and MediaPlayer allows you to resume playback after pausing by calling Play(), the Storyboard doesn't work in the same way. Instead, a separate ResumeStoryboard action is required. If this isn't the behavior you want, you can consider adding some code for your play button instead of using the declarative approach.

■ **Note** The downloadable code samples for this chapter include a declarative media player window and a more flexible code-driven media player window.

Playing Multiple Sounds

Although the previous example showed you how to control the playback of a single media file, there's no reason you can't extend it to play multiple audio files. The following example includes two buttons, each of which plays its own sound. When the button is clicked, a new Storyboard is created, with a new MediaTimeline, which is used to play a different audio file through the same MediaElement.

```
<Grid>
 <Grid.RowDefinitions>
   <RowDefinition Size="Auto"></RowDefinition>
   <RowDefinition Size="Auto"></RowDefinition>
  </Grid.RowDefinitions>
  <MediaElement x:Name="media"></MediaElement>

  <Button>
    <Button.Content>Click me to hear a sound.</Button.Content>
    <Button.Triggers>
      <EventTrigger RoutedEvent="Button.Click">
        <EventTrigger.Actions>
        <BeginStoryboard>
          <Storyboard>
            <MediaTimeline Source="soundA.wav"
             Storyboard.TargetName="media"></MediaTimeline>
          </Storyboard>
        </BeginStoryboard>
        </EventTrigger.Actions>
      </EventTrigger>
    </Button.Triggers>
  </Button>

  <Button Grid.Row="1">
    <Button.Content >Click me to hear a different sound.</Button.Content>
    <Button.Triggers>
      <EventTrigger RoutedEvent="Button.Click">
        <EventTrigger.Actions>
          <BeginStoryboard>
            <Storyboard>
              <MediaTimeline Source="soundB.wav"
               Storyboard.TargetName="media"></MediaTimeline>
            </Storyboard>
          </BeginStoryboard>
        </EventTrigger.Actions>
      </EventTrigger>
    </Button.Triggers>
  </Button>
</Grid>
```

In this example, if you click both buttons in quick succession, you'll see that the second sound interrupts the playback of the first. This is a consequence of using the same MediaElement for both timelines. A slicker (but more resource-heavy) approach is to use a separate MediaElement for each button and point the MediaTimeline to the corresponding MediaElement. (In this case, you can specify the Source directly in the MediaElement tag, because it doesn't change.) Now, if you click both buttons in quick succession, both sounds will play at the same time.

The same applies to the MediaPlayer class—if you want to play multiple audio files, you need multiple MediaPlayer objects. If you decide to use the MediaPlayer or MediaElement with code, you have the opportunity to use more intelligent optimization that allows exactly two simultaneous sounds, but no more. The basic technique is to define two MediaPlayer objects and flip between them each time you play a new sound. (You can keep track of which object you used last using a Boolean variable.) To make this technique really effortless, you can store the audio file names in the Tag property of the appropriate element, so all your event handling code needs to do is find the right MediaPlayer to use, set its Source property, and call its Play() method.

Changing Volume, Balance, Speed, and Position

The MediaElement exposes the same properties as the MediaPlayer (detailed in Table 26-1) for controlling the volume, the balance, the speed, and the current position in the media file. Figure 26-2 shows a simple window that extends the sound player example from Figure 26-1 with additional controls for adjusting these details.

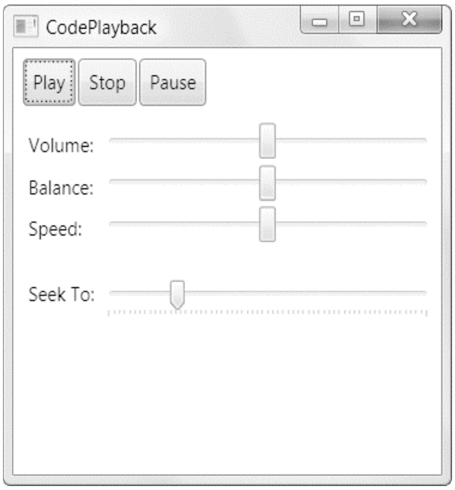

Figure 26-2. *Controlling more playback details*

The volume and balance sliders are the easiest to wire up. Because Volume and Balance are dependency properties, you can connect the slider to the MediaElement with a two-way binding expression. Here's what you need:

```
<Slider Grid.Row="1" Minimum="0" Maximum="1"
 Value="{Binding ElementName=media, Path=Volume, Mode=TwoWay}"></Slider>
<Slider Grid.Row="2" Minimum="-1" Maximum="1"
 Value="{Binding ElementName=media, Path=Balance, Mode=TwoWay}"></Slider>
```

Although two-way data binding expressions incur slightly more overhead, they ensure that if the MediaElement properties are changed some other way, the slider controls remain synchronized.

The SpeedRatio property can be connected in the same way:

```
<Slider Grid.Row="3" Minimum="0" Maximum="2"
  Value="{Binding ElementName=media, Path=SpeedRatio}"></Slider>
```

However, this has a few quirks. First, SpeedRatio isn't used in a clock-driven audio (one that uses a MediaTimeline). To use it, you need to set the LoadedBehavior property of SpeedRatio to Manual and take control of its playback manually through the playback methods.

■ **Tip** If you're using a MediaTimeline, you can get the same effect from the SetStoryboardSpeedRatio action as you get from setting the MediaElement.SpeedRatio property. You learned about these details in Chapter 15.

Second, SpeedRatio isn't a dependency property, and WPF doesn't receive change notifications when it's modified. That means if you include code that modifies the SpeedRatio property, the slider won't be updated accordingly. (One workaround is to modify the slider in your code, rather than modify the MediaElement directly.)

■ **Note** Changing the playback speed of audio can distort the audio and cause sound artifacts, such as echoes.

The last detail is the current position, which is provided by the Position property. Once again, the MediaElement needs to be in Manual mode before you can set the Position property, which means you can't use the MediaTimeline. (If you're using a MediaTimeline, consider using the BeginStoryboard action with an Offset to the position you want, as described in Chapter 15.)

To make this work, you don't use any data binding in the slider:

```
<Slider Minimum="0" Name="sliderPosition"
  ValueChanged="sliderPosition_ValueChanged"></Slider>
```

You use code like this to set up the position slider when you open a media file:

```
private void media_MediaOpened(object sender, RoutedEventArgs e)
{
    sliderPosition.Maximum = media.NaturalDuration.TimeSpan.TotalSeconds;
}
```

You can then jump to a specific position when the slider tab is moved:

```
private void sliderPosition_ValueChanged(object sender, RoutedEventArgs e)
{
    // Pausing the player before moving it reduces audio "glitches"
    // when the value changes several times in quick succession.
    media.Pause();
```