Accessibility for Everyone: Understanding the Section 508 Accessibility Requirements

JOHN MUELLER

a!™
Apress™

Accessibility for Everyone: Understanding the Section 508 Accessibility Requirements
Copyright © 2003 by John Mueller

The source code for this book is available to readers at http://www.apress.com in the Downloads section.

Using Microsoft Active Accessibility

In This Chapter:

- Why Should You Care About MSAA?
- How Can You Use the `AccessibleObject` Class to Your Advantage?
- What Are the Standard Accessibility Options?
- How Can the Standard Accessibility Options Help You as a Developer?
- How Can You Access the Keyboard Features?
- How Do You Display the Windows Accessibility Feature Status Information?
- How Can You Interact with the Sound Features?
- How Can You Detect the Display Features?
- How Can You Detect the Mouse Settings and Functionality?

WINDOWS COMES WITH support for accessibility built into the system. In fact, this support, when it works, provides many of the features that anyone with special needs would require for a desktop application. The problem is that many developers don't include the required support in their applications, so Windows users don't gain full access to the accessibility features this support could provide. Of course, the other side of the coin is that while Windows does provide good support for accessibility functionality, the topic of accessibility doesn't exactly head the list of developer conference topics. Consequently, the main purpose of this chapter is to make you aware of what accessibility features are available and demonstration how you can use them in your next application.

NOTE   Many developers recognize that Windows provides one of the few operating system platforms where it's possible to add a level of accessibility support to an application without a lot of added coding. In this respect, Windows does lead the world in providing the means for those with special needs to help themselves. In fact, Microsoft is helping in other ways. A recent Canadian Nation Institute for the Blind (CNIB) article (`http://www.cnib.ca/Thatallmayread/news_release.htm`) discusses the contributions that Microsoft has made to the well-being of those with special needs.

The first part of this chapter discusses the Windows Accessibility features from a user perspective. I've talked with over a hundred developers during the writing of this book (and I have over half of the book to write yet). Out of all of those developers, only one had even heard that Windows has Accessibility features and understood how to install and use them. If you already know about the Windows Accessibility features, you can safely skip the first section of the chapter. On the other hand, if you don't know what Windows can provide, this first section will help you understand the features we'll use in the applications in this chapter.

The next several sections discuss each of the Accessibility features in turn. We begin by discussing the keyboard features, then we move on to sound, then the display, and, finally, the mouse. When you finish these sections, you'll see just how complete the Accessibility features are so long as you provide the required support in your application. Of course, the Accessibility features have some support holes that we'll discuss as the chapter progresses. You can plug some of these holes by adding support for special hardware.

The final section of the chapter discusses an essential topic. You need to know when a user has turned on the Accessibility features for a particular machine. More than that, you need to know how the user has configured accessibility support so that your application can work with the Accessibility features, rather than against them. This is especially important for some of the visual and audio features because they require special coding in your application.

What Is Microsoft Active Accessibility (MSAA)?

Microsoft Active Accessibility (MSAA) is a set of COM classes that help you build better accessible applications. These COM classes help you create a bond between the application and the operating system. This bond enables the operating system to perform tasks such as querying the application for additional help information and asking the application to perform specific tasks.

ON THE WEB You'll find MSAA general information, along with downloads used in this book, on the Microsoft Active Accessibility Web site at `http://msdn.microsoft.com/library/default.asp?url=/nhp/Default.asp?contentid=28000544`. This Web site includes links to a number of helpful articles and other resources. Fortunately, MSAA is built into Windows XP and you can add it to other versions of Windows through service packs (including Windows 98, but not Windows 95). If you need to support Windows 95 users, then be sure to download the MSAA SDK at `http://download.microsoft.com/download/activaxs/SDK/1.3/W95/EN-US/MSAA13SDK.exe`. We'll use a number of MSAA specific tools in this book, so you'll want to download them from `http://www.microsoft.com/downloads/release.asp?ReleaseID=33491`.

However, as with most products, the COM interface is just the tip of the iceberg. You'll find that MSAA also supports registry entries that you can monitor and a few API functions you can use to perform specific tasks. In general, .NET users will find that Microsoft has built the major MSAA functionality into the .NET Framework, but that some ancillary features don't exist yet. Users of earlier Visual Studio product versions will need to work a little harder to gain access to MSAA features.

The reason that the MSAA section appears in this chapter is because it's important for the developer to know a little about the underlying technology before looking at the user interface elements. However, understanding the user interface elements and learning how they work is an important part of working with MSAA—you can't test an MSAA application otherwise. Consequently, we'll discuss the MSAA theory in this chapter before we discuss the user interface.

The sections that follow perform two tasks. First, you'll learn how MSAA works from both an operating system and a development platform perspective. Second, you'll get a quick demonstration of how MSAA works. We'll explorer MSAA programming methods in detail in the "Obtaining and Using Microsoft Active Accessibility" sections of Chapter 7.

Understanding the Technical Details

Working with MSAA requires that you understand a number of technical details. The first is that most of the functionality needed to use MSAA appears in the OLEACC.DLL. It's interesting to open this DLL up using the Depends utility because you can see many of the API calls directly and learn the dependencies of this DLL. Figure 6-1 shows the OLEACC.DLL file opened in Depends (also known as the Dependency Walker as shown in this screenshot). Notice the list of API functions, such as `AccessibleChildren()` in the exports list. Scroll through the list and you'll find essential interface references such as `IID_IAccessible`.
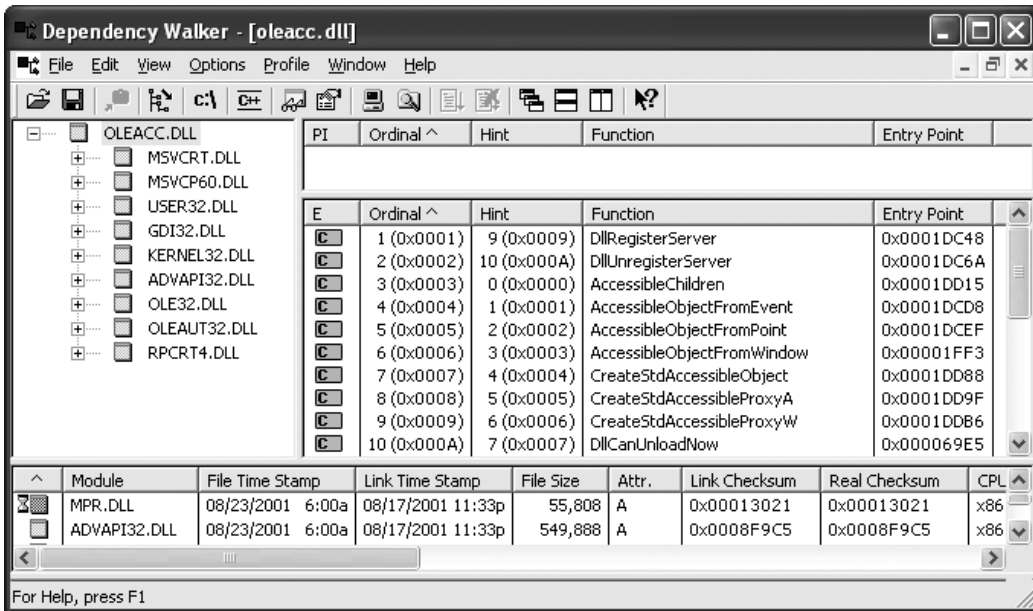
Figure 6-1. The OLEACC.DLL file provides a list of essential API calls, as well as COM functionality.

A second technical detail is that MSAA is a COM object. The interface in question is `IAccessible`. The .NET Framework wraps the more important parts of the COM functionality found in `IAccessible` in the `AccessibleObject` class. The `IAccessible` interface provides the methods that make it possible to request information about a component or control, such as a description. This interface also provides access to functionality, such as getting the current focus or performing the default action. In short, the `IAccessible` interface encapsulates most of the programming functionality that a developer needs to interact with the accessibility features of any control.

ON THE WEB   You'll find a good summary of the `IAccessible` functions at `http://msdn.microsoft.com/library/en-us/msaa/msaaccrf_4f51.asp`. Note especially the descriptive properties and methods that this interface provides. Another good place to get an overview of the `IAccessible` interface is at `http://msdn.microsoft.com/library/en-us/msaa/msaaccrf_5q05.asp`.

A third technical detail is that MSAA relies to an extent on messaging—the same technique used by a number of other processes in Windows. The OLEACC.DLL

uses the `WM_GETOBJECT` message to retrieve an Active Accessibility server application object. In other words, if you create an MSAA application, you'll need to handle this message in some cases.

A fourth technical detail is the registry entries. You can monitor the registry to obtain the current accessibility settings for Windows. The key your application would need to monitor is HKEY_CURRENT_USER\Control Panel\Accessibility. Figure 6-2 shows the Windows XP registry setup. Older versions of Windows are slightly less robust than Windows XP. For example, Windows 2000 lacks the Blind Access key shown in Figure 6-2. However, you can count on all versions of Windows to support accessibility function keys such as HighContrast, SerialKeys, and MouseKeys. We'll discuss the monitoring process in the "Developing the Windows Accessibility Status Application" section of this chapter.
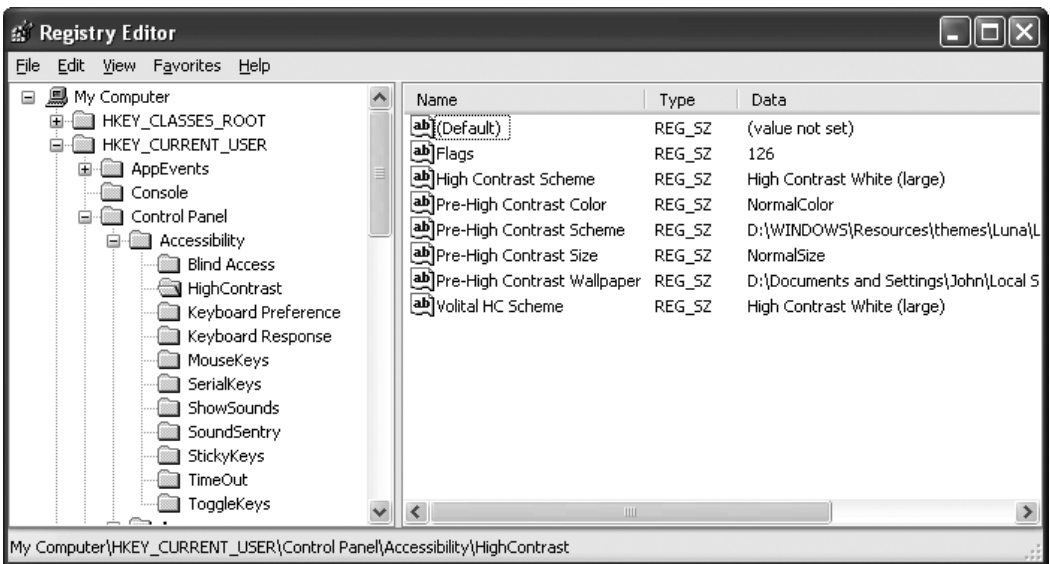
Figure 6-2. Use the registry to determine the status of the various Windows Accessibility options.

The four technical details help you understand what's going on beneath the surface. You'll learn more about the linkage between these technical details and accessibility in general as the chapter progresses. For now, what you need to know is that there's linkage between MSAA and Windows Accessibility. The user interface features the user selects affect messaging, the registry, and the COM interface.

A Quick Demonstration of the AccessibleObject Class

As with many of the other examples we'll discuss in this book, you'll need to add a reference to the Accessibility.DLL that comes with the .NET Framework. When you add this reference, you can view it in the Object Browser. Figure 6-3 shows a typical display of this library. Notice that it contains references to both IAccessible and IAccessibleHandler. In addition, you have access to the majority of the functions found in the OLEACC.DLL file. You also need this reference to gain access to the AccessibleObject class.
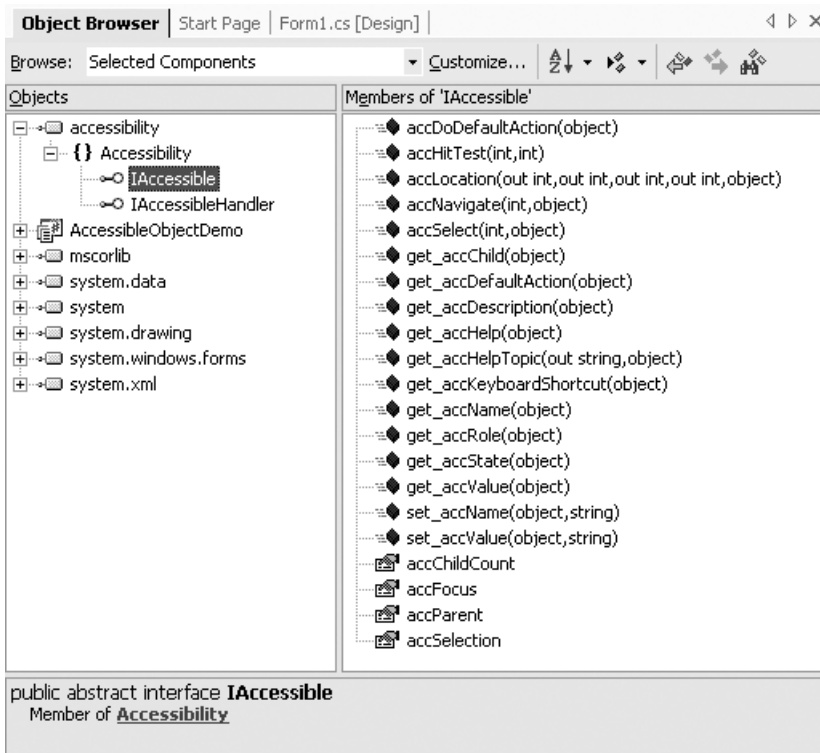
Figure 6-3. The Accessibility.DLL file contains the two interfaces required for the COM portion of an accessible application.

Developers can use the AccessibleObject class in a number of ways. For example, you can use it to create an accessible version of a new component or control. By adding this class to your component or control, and overriding the default actions it provides, you can customize the accessibility information the

component or control provides to the user. In many cases, the customization makes it much easier for the user to interact with your application as a whole.

A more common use of the `AccessibleObject` class is to gain complete access to the accessibility features provided by existing components and controls. For example, you'll need it to gain access to the shortcut key for some types of controls. It also provides a number of handy methods such as `DoDefaultAction()`, which performs the default action supported by any accessible object. The demonstration application relies on the features of the `AccessibleObject` class to create a specialized tooltip presentation for the user. Listing 6-1 shows this example. You'll find the complete listing in the \Chapter 06\AccessibleObjectDemo folder of the source code that you can obtain from the Downloads section of the Apress Web site (`http://www.apress.com`).

Listing 6-1. Specialized `ToolTip` *Code for Displaying Complete Accesibility Information*

```
public frmMain()
{
   // Required for Windows Form Designer support
   InitializeComponent();

   // Initialize the accessible objects.
   btnQuit.AccessibleDefaultActionDescription =
      "Press to exit the application.";
   btnTest.AccessibleDefaultActionDescription =
      "Press to test the application.";
   txtMessage.AccessibleDefaultActionDescription =
      "Type to change test message.";
}

private void SpecialTip(object sender, System.EventArgs e)
{
   Control         Ctrl;   // The control in question.
   AccessibleObject AO;     // The accessibility information.
   ToolTip          TT;     // Special ToolTip
   StringBuilder    Output; // ToolTip Output String.

   // Initialize the ToolTip.
   TT = new ToolTip();
   TT.AutoPopDelay = 7000;
   TT.AutomaticDelay = 300;
```

```
      // Get the sender information.
      Ctrl = (Control)sender;

      // Obtain access to the accessibility information.
      AO = Ctrl.AccessibilityObject;

      // Create the output string.
      Output = new StringBuilder();
      Output.Append("Name: ");
      Output.Append(AO.Name);
      Output.Append("\r\nRole: ");
      Output.Append(AO.Role);
      Output.Append("\r\nDescription: ");
      Output.Append(AO.Description);
      Output.Append("\r\nDefault Action: ");
      if (AO.DefaultAction == null)
         Output.Append("None");
      else
         Output.Append(AO.DefaultAction);
      Output.Append("\r\nKeyboard Shortcut: ");
      if (AO.KeyboardShortcut == null)
         Output.Append("None");
      else
         Output.Append(AO.KeyboardShortcut);
      Output.Append("\r\nState: ");
      Output.Append(AO.State);
      Output.Append("\r\nValue: ");
      if (AO.Value == null)
         Output.Append("None");
      else
         Output.Append(AO.Value);

      // Display the information on screen.
      TT.SetToolTip(Ctrl, Output.ToString());
      TT.Active = true;
}
```

It's important to create a customized `AccessibleDefaultActionDescription` property value for each component in your application. The reason is simple. The default information simply tells the user to press the spacebar to perform the default action, but it doesn't say what that action is. In some cases, such as a text box, the control doesn't have any type of default action assigned to it. All the user knows is that the text box exists. Telling the user they can type something might

seem obvious until you try to use the example without the benefit of seeing it. In fact, you should try this application out by blindfolding yourself and using just the Narrator application described later in the chapter to move from area to area.

The SpecialTip() method has to follow the setup for the MouseHover delegate so that it can act as an event handler. Consequently, it lists the sender as an object, and not a control as you might expect. You need to add this event handler to the MouseHover event of every control on the form, as shown in Figure 6-4.
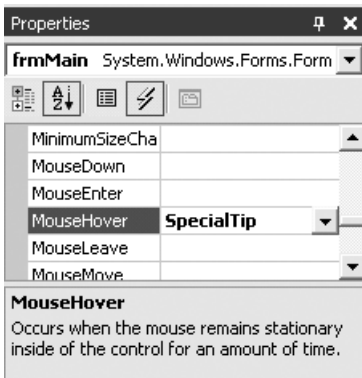
Figure 6-4. Use the MouseHover *event to create linkage between the user action and the* SpecialTip() *event handler.*

The SpecialTip() method code begins by creating the ToolTip object, TT. Notice that TT uses an AutoPopDelay value of 7000 ms and an AutomaticDelay value of 300 ms to ensure proper operation. A few tests will show you that the changes are needed to ensure that the tooltip actually pops up fast enough for someone with mobility difficulties and stays up long enough for someone with cognitive difficulties to read. Interestingly enough, most screen readers will continue saying the text in the tooltip even after it disappears from view provided the user doesn't change focus.

After the code creates the tooltip, it gains access to the sender as a Control object. It uses the Control object's AccessibilityObject property to create the AccessibleObject, AO. Finally, the code uses AO to fill out the entries in a StringBuilder object, Output. At this point, we have a string that contains all of the essential accessibility information provided by the control that activated the SpecialTip() event handler. The final step is to place this information in TT using the SetToolTip() method, and then display the tooltip by setting the Active property to true. Figure 6-5 shows the output of this application.
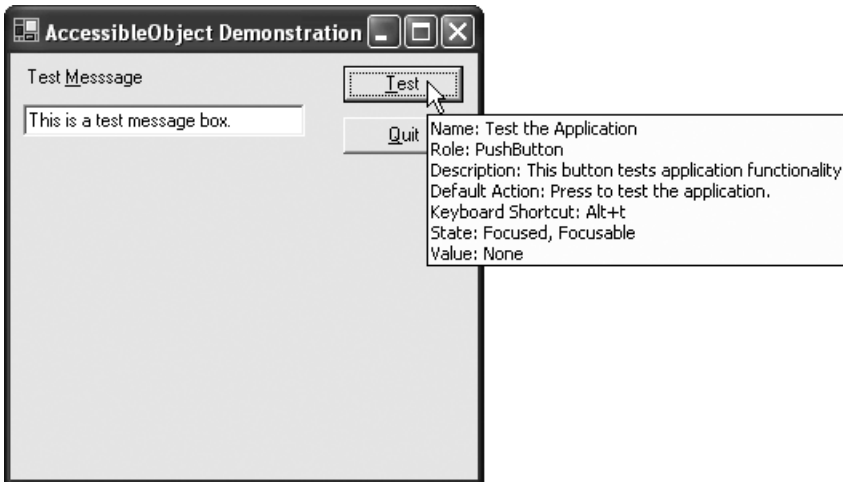
Figure 6-5. The SpecialTip() *event handler outputs tooltips with complete accessibility information.*

The output shown in Figure 6-5 is probably a little too inclusive. However, you could provide a menu option that helps the user customize this information. Some users, especially those who use screen readers, will probably want the keyboard shortcut information as part of the tooltip. Users with cognitive needs might find it helpful to set the tooltip popup and display times individually. Most users will want to see the description—using this technique means you don't have to define that part of the tooltip for every control. Although this example shows what's available, you'll still want to augment the code to provide added flexibility.

An Overview of the Standard Accessibility Options

Windows actually provides a wealth of accessibility features, but many people don't realize they even exist. You can divide the accessibility features into two areas. The first area is utilities that you run as needed. The second are services that you control using the Accessibility applet in the Control Panel. Third party companies probably don't have much to worry about from these utilities, but they do work well enough for many people, including developers of accessible applications.

TIP You can often find other aids in other Control Panel applets. For example, look in the Keyboard applet and you'll notice options that affect the character repeat delay and the character repeat rate. The Mouse applet contains settings that enable the user to adjust the double click rate and turn on features such as ClickLock. The ClickLock feature is especially interesting because it helps someone drag and drop items without holding the mouse button down. The Mouse applet also provides access to larger pointers and enables the user to switch mouse button functionality. Try using your application with changes to these mouse and keyboard settings in place to see how it reacts. This test can help ensure that time-dependent features work as anticipated.

The following sections provide a user eye view of accessibility in Windows. This material is important because you need to know how the user will view the output of your application. However, the utilities and services described in these sections represent a worst-case scenario for many accessible application users. Third party vendors have developed software and hardware that provide superior performance and usability. In short, if your application works well with this software, it will likely work well with the high-end products that most accessible application users will have installed on their systems.

Utilities

Utilities are applications that the user can run from the Start ➤ Programs ➤ Accessories ➤ Accessibility menu if this feature is installed on the host system. Windows XP installs the feature by default, so most users of that operating system will have this feature at their fingertips by default. Users of older versions of Windows can install Windows Accessibility using the Add/Remove Programs applet in the Control Panel.

ON THE WEB   Sometimes a developer will need short-term changes in screen appearance to accommodate environmental conditions (such as the change from daylight to nighttime lighting conditions). NightVision (`http://www.adpartnership.net/NightVision/index.html`) is a free utility that helps the developer make these changes. It helps by modifying the gamma (or brightness) values for the display. The interesting feature of this utility is that you can create your own settings. For example, I found it useful as an aid to seeing the screen as someone with low vision would see it. Using this product can help you make color and brightness choices and keep the display readable even in bright light or low vision conditions.

Several of these utilities actually have interesting uses for developers outside of their use for accessibility development. We'll explore these utilities and their interesting uses in the sections that follow. Make sure you try out each of the utilities as you read about them and then spend time working with them afterward. I've personally found many of the utilities useful as programming aids.

Using the Magnifier

The Magnifier is equivalent to a magnifying lens used for reading. Whenever you start the Magnifier, you'll see a band open at the top of your screen. This band contains a magnified version of the information contained at the mouse cursor. You can change the size of the band by dragging the line separating it from the rest of your display using the mouse, just as you would for the Taskbar.

From a developer perspective, you can use this tool to see how your application reacts to the unexpected. Suddenly cutting off the upper half of the screen is a good way to see what will happen when someone needs that portion of the display for an accessibility need such as a keyboard. The Magnifier also helps you view the fit and finish of your application. For example, when an application displays a logo or other picture element, it's sometimes difficult to determine if the placement on screen is correct.

As previously mentioned, the Magnifier works on the same basis as a lens used for reading. However, this lens is adjustable, making it possible to see the screen at various levels of detail. The default magnifier setting will magnify items by a factor of 2. You can change this setting using the Magnifier Settings dialog box shown in Figure 6-6. This dialog box opens whenever you start the Magnifier.
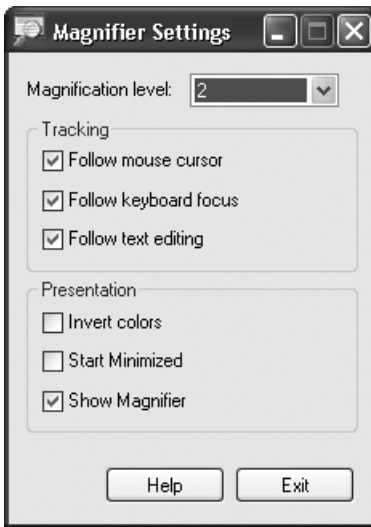
Figure 6-6. The Magnifier Settings dialog box allows you to control how the Magnifier interacts with your system.

Notice you can change the tracking technique that Magnifier uses to home in on the action. The default setting tells Magnifier to follow any activity on the screen. However, you can clear one or more of these check boxes to reduce the level of activity. When I use the Magnifier to check my application for fit and finish problems, I normally tell it to follow the mouse. On the other hand, when checking the application for usage problems, it's better to have the Magnifier follow the keyboard input. That way you can simply tab between fields on a form as needed.

TIP   As a developer and someone interested in drawing, I often use Magnifier to see how other people create small screen elements such as icons. It's hard to get just the right color combinations without help at times. Viewing someone else's work can mean the difference between trial and error, and getting the drawing right the first time.

You'll also find three check boxes in the Presentation section. The first tells Magnifier to invert the screen colors. This makes it easier to contrast the normal appearance of your display with the magnified version. I also find this an interesting

way to check the color settings of my application—it should display well in either mode. You can also start the Magnifier Settings dialog box minimized. Finally, you can clear Show Magnifier to get rid of the band at the top. This is a handy feature if you need more screen real estate for a short time and don't want to disable Magnifier in the interim.

Magnifier does incur a noticeable performance hit on your system, so you'll want to turn it off when you performance tune your application. When you click Exit in the Magnifier Settings dialog box, the Magnifier also stops working and the display returns to normal.

Using the Narrator

The Narrator reads the essential content on screen to you. In addition, whenever it encounters a control, it will provide information about that control. The information provided depends on the developer of the application. In many cases, there's a generic, almost useless bit of text that Narrator will read about the control if the developer provides no other information.

The most obvious use of this utility for the developer is to check the accessibility information provided by the application. Using a screen reader of some type is the only effective means of performing this task. This application acts as a sanity check for both desktop and Web-based applications. A developer can listen to hear if Narrator stumbles on the content. If it doesn't, then no one else is likely to have troubles either.

There are other good uses for screen readers from the developer perspective. I find that it does a relatively good job and use it when I need to "read" documents online. I'll get the document placed on screen, then kick back and let Narrator do the work while I concentrate on the information contained on the Web site.

ON THE WEB   AWS, a screen reader from Freedom Scientific (`http://www.freedomscientific.com/`), provides a much better environment in which to test your screen reader–capable applications. Not only can you choose from a variety of voices, but you can also select options such as the voice pitch and reading speed with greater accuracy than Narrator can. JAWS also provides Braille support, along with a wealth of other features. You can download a demonstration version of JAWS at `http://www.freedomscientific.com/fs_downloads/jaws.asp`. The demonstration version provides most of the features of the full product, but you can only use it for about 30 minutes. The Help menu provides an estimate of the time you have left for using the demonstration version.

Sometimes it's also important to use Narrator as a fit and finish tool. For example, closing your eyes and listening to the prompts for an application can help you locate grammar and spelling problems. You can also hear the prompts and determine if they make sense—whether they're consistent and easy to understand. The Narrator dialog box shown in Figure 6-7 contains four options.
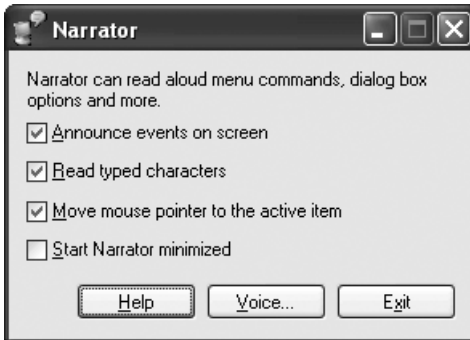
Figure 6-7. Use the Narrator dialog box settings to control how Narrator interacts with your system.

Announce Events On Screen: Tells Narrator to announce when you successfully complete an action such as changing windows. (A window change can also include the appearance of message boxes and other system events that the user didn't cause.)

Read Typed Characters: Tells you which character you typed last, including control characters such as backspace.

Move Mouse To Active Item: Moves the mouse cursor so that you can see which item on screen has the focus.

Start Narrator Minimized: Starts the program with the Narrator dialog box minimized.

The Voice button takes you to the Voice Settings dialog box. This dialog box allows you to choose a new voice for Narrator. The default setting is Microsoft Sam. You can also change the speed, volume, and pitch of the voice. Nothing you do will make the Narrator sound completely human. However, adjusting the pitch and speed does make Narrator friendlier. Adjust the pitch and volume settings to make Narrator easier to understand.

Using the On-Screen Keyboard

The On-Screen Keyboard shown in Figure 6-8 is a replica of the keyboard attached physically to your system. It allows you to type text using a mouse or other pointing device. We initially discussed this accessibility feature in the "Evaluating Your Audience" section of Chapter 4. You'll want to review that section for some of the interesting uses of the On-Screen Keyboard.

Figure 6-8. You can use the On-Screen Keyboard as an alternative means for inputting data.

You can adjust the appearance of the keyboard using the options on the Keyboard menu. For example, you might want to use a 106 key keyboard instead of the 101 key default. You can also choose between a standard and an enhanced keyboard. As mentioned in Chapter 4, many developers find the block layout easier to use because the keys are lined up and easier to access. The staggered layout used by physical keyboards is fine when you need to type, but it can prove cumbersome for mouse or joystick access.

The Settings menu controls how the On-Screen Keyboard interacts with the system. For example, you might want to use a different font for the keys. You can also choose whether the keyboard always remains on top. The Use Click Sound option comes in handy if you're used to a regular keyboard and miss the sound it makes.

The Settings ➤ Typing Mode command displays the Typing Mode dialog box. You'll use the options on this dialog box to control how the user inputs data. You have a choice between clicking, hovering, or using a joystick (or other recognized device). To get a better feel for how frustrating it can be to use an eye gaze system, set the keyboard up for hover mode. You'll find that using a joystick or mouse to select a key, leave it in place for the required time, and then move to the next key is harder than it first appears.

Using the Accessibility Wizard

You'll find the Accessibility Wizard on the Start ➤ Programs ➤ Accessories ➤ Accessibility menu. The main purpose for using this wizard is to set Windows up for an individual's accessibility needs. The user isn't required to use the wizard, but it does make things easier if the user has many changes to make and is not familiar with the actual settings. Once you start the Accessibility Wizard, you'll see a Welcome dialog box and then a series of question dialog boxes like the one shown in Figure 6-9. Just follow the prompts to configure accessibility support to meet a particular need.
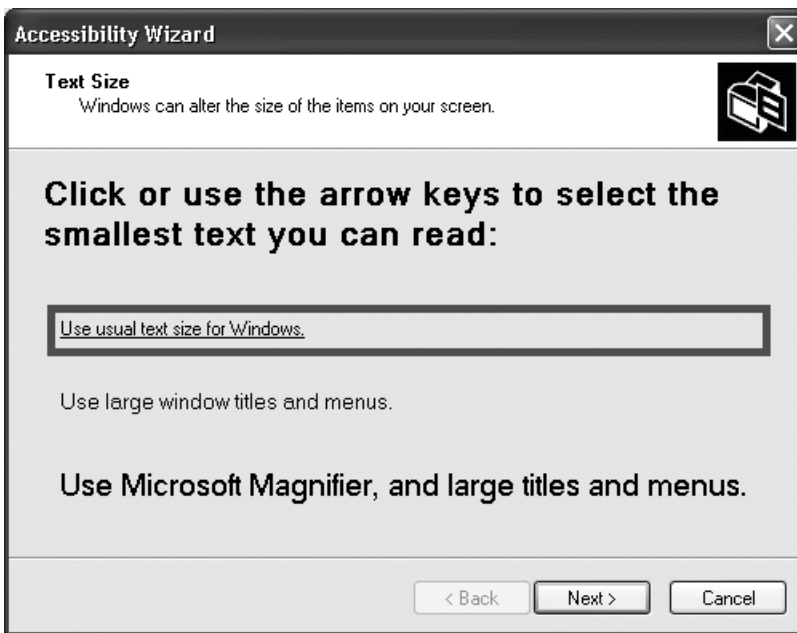
Figure 6-9. Running the Accessiblity Wizard will help a user configure Windows Accessibility support to meet particular needs.

ON THE WEB   If you find that you need additional help with the Accessibility Wizard, you'll find step-by-step instructions for the various setups at `http://www.microsoft.com/enable/training/windowsxp/usingwizard.htm`. The instructions are targeted at a specific type of setup, so you'll need to decide which of the procedures to follow.

For the developer, the Accessibility Wizard provides a fast means of configuring a test system to meet specific needs. In addition, the Accessibility Wizard shows one technique for providing special needs configuration of your own applications. Although the wizard provided with your application doesn't have to be this complex, the simple question and answer format does make the application a lot easier to configure at the outset. Another product that uses this question and answer format is the JAWS installation program (see the "Using the Narrator" section for details).

Using the Utility Manager

So far, this book has spent a lot of time promoting the advantages of flexible application configuration. The Utility Manager is a special application for managing the Windows Accessibility applications. It controls the Magnifier, On-Screen Keyboard, and Narrator. All three of these applications appear in the field at the top of the dialog box (along with any other Accessibility applications you may install), as shown in Figure 6-10. The Utility Manager demonstrates that you can make an application flexible, yet provide a central configuration point for it.

Figure 6-10. Manage your Windows Accessibility applications using the Utility Manager shown here.

As you can see, the Utility Manager displays the status of each of the applications that it manages. To stop an application, highlight it and click Stop. Likewise,

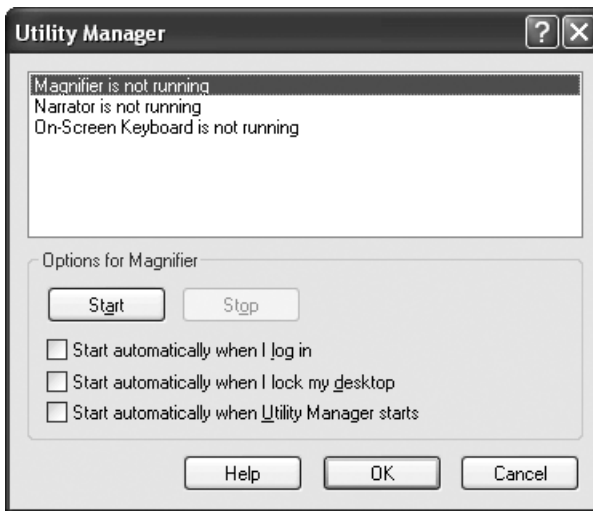click Start to restart the application. Notice that most, but not all of the keys have shortcuts—any management application you create should include shortcuts for all options. In addition, the Utility Manager lacks support for tooltips—something that most users find helpful in determining what action to perform. (Interestingly enough, you can right click an entry or press the Context Menu key, and then select What's This from the context menu to learn more about a particular option.)

You can also set the startup options for each application. The three options will start the program when you log in, lock the system, or start the Utility Manager when they're checked. Again, configuration flexibility is extremely important. Enabling the user to choose when the application will start makes the application itself seem flexible.

From a developer perspective, the Utility Manager provides a quick means of starting and stopping Windows Accessibility applications. Sometimes you need quick access for testing because you might want the feature on during debugging and off while you correct code.

Accessibility Options Applet

The Accessibility Options applet appears in the Control Panel. It helps you configure the Windows Accessibility options individually. Using this applet helps you learn about each option individually and see how the Accessibility Wizard combines them to meet specific special needs. Through observation, you can learn a great deal about how the various accessibility options work under Windows and develop your applications accordingly.

TIP   There's a chance that constantly turning the Windows Accessibility options on and off will cause certain elements of your display, especially the fonts and icons, to become large and stay that way. If this happens, turn off all accessibility features and restart the machine. Open the Themes tab of the Display Properties dialog box and choose the Windows XP theme. Click Apply. The text and icons should appear normal again. Obviously, this fix relies on an unaltered Windows XP theme, so you'll want to ensure that you never make changes to it. Always make any theme changes to another file.

The following sections discuss each of the Windows Accessibility options. Make sure you try each of the options to learn how they work. More importantly, use the options as part of your application testing strategy. The Windows Accessibility options should be part of the manual usability test you perform on every application your organization develops.

Using the StickyKeys Feature

The StickyKeys feature is one of three options on the Keyboard tab of the Accessibility Options dialog box shown in Figure 6-11. It's useful for a variety of purposes. For example, this option forces the Shift, Ctrl, and Alt keys to act as toggle switches. Press one of these keys once and it becomes active; press it a second time and it's turned off. The user can also press the keys sequentially to activate them or press a non-control key to execute the entire key press (such as Ctrl+Alt+A). Afterward, the control keys automatically become inactive. In other words, the toggle feature is only important if the user doesn't want to use that key as part of a key press.
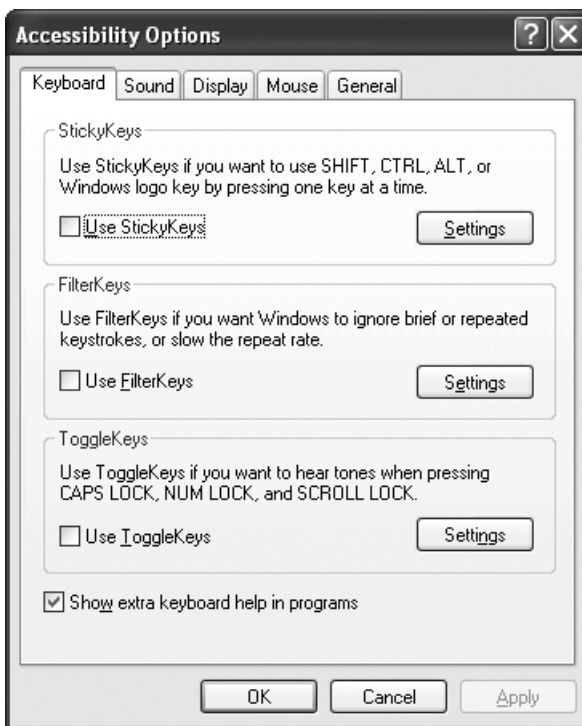
Figure 6-11. Use the keyboard options to modify the functionality of the PC keyboard.

From a developer perspective, there are a number of good uses for the StickyKeys feature. For example, you can use StickyKeys in graphics programs that require you to hold down the Ctrl key to select a group of items. It can be inconvenient to hold down the Ctrl key while you look around for objects to select. The StickyKeys feature alleviates this problem.

At some point, you'll want to configure StickyKeys to meet particular needs. Click the Settings button on the Keyboard tab of the Accessibility Properties dialog box to open the Settings for StickyKeys dialog box shown in Figure 6-12. As you can see, the dialog contains a number of optional settings that enable StickyKeys to react to user input in a variety of ways.
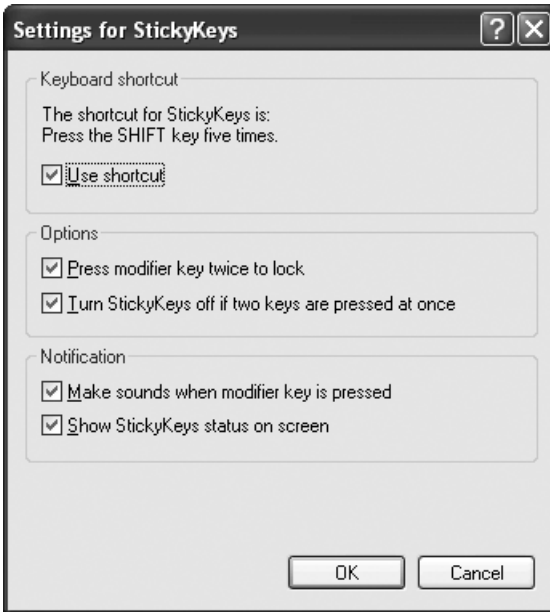
Figure 6-12. Configure the StickKeys options using this dialog box.

There are three groups of settings for StickyKeys. The first group, Keyboard Shortcut, enables you to turn on StickyKeys using the shortcut key. There's no reason to turn this off. It's very unlikely that another application would use the same control key sequence. Actually, it's a little surprising that Microsoft chose such an odd key combination given the reason they provide this feature. We'll discuss an easier technique for turning StickyKeys on and off in the "Developing the Windows Accessibility Status Application" section of the chapter.

The Options group contains two settings. The StickyKeys option usually works like a toggle. Checking the first box tells Windows to wait until you press the same control key twice before making the control key active. The second check box enables two people to use the same keyboard if one needs to use StickyKeys and the other doesn't. Pressing a control key and a non-control key at the same time turns StickyKeys off in multiuser environments where this feature could be disorienting.

The Notification group also contains two settings. The first setting tells Windows to play a different sound for each unique control key it activates. This setting can help prevent you from activating a control key by accident. The second option displays an icon on the Taskbar so that you can control StickyKeys more easily. Select this option to make it easier to turn StickyKeys on and off.

Using the FilterKeys Feature

Many users experience problems typing keys correctly—they sometimes tap the key twice when they really meant to tap it only once. In other cases, the user will press the key too long because they don't have the sense of touch and control. FilterKeys helps eliminate extra keystrokes so you don't get "tthis" instead of "this." It performs this task by setting a minimal time threshold between keystrokes. As with StickyKeys, you can adjust the way FilterKeys works by clicking the associated Settings button. Figure 6-13 shows the Settings for FilterKeys dialog box.

Figure 6-13. Define features such as the timespan between keystrokes using this dialog box.

The first option in this dialog box enables you to turn the shortcut key option on or off. This works just like the same feature in StickyKeys. In fact, you'll find that all of the Windows Accessibility features provide this option, so I'll skip it in future discussions. I also found Microsoft's selection of a shortcut for this feature a bit odd considering how the feature is used.

The Filter Options group provides options to let the user choose between two methods of filtering keystrokes. The first option filters keys that the user presses in rapid succession. This feature would filter the rapid typing of the extra "t" in the previous example. The Settings button displays a dialog box that enables you to select how long an interval must pass between the first and second times you press the same key. It also provides a field in which you can test the setting.

The second option in the Filter Options group filters accidental key presses. At one time or another, everyone presses a key without meaning to. As with the StickyKeys option, the Settings button displays a dialog box in which you select how long you have to press a key before Windows accepts it. This dialog box also lets you change the actual repeat rate or turn the keyboard repeat feature off so that the user must press each key individually.

The Notification group at the bottom of the dialog box should look familiar. The only difference is that FilterKeys beeps when you activate it instead of playing a sound (such as a WAV file). You can also display an indicator on screen to show that FilterKeys is active.

Using the ToggleKeys Feature

How many times have you accidentally hit the Caps Lock key and found yourself typing in all uppercase? I know that when I get busy, it does occasionally happen to me. The ToggleKeys feature emits a tone every time you turn the Caps Lock, Scroll Lock, or Num Lock key on or off. This feature enables you to detect toggle key changes quickly, before you've typed a lot of material using the wrong case.

Using the SoundSentry Feature

The SoundSentry and ShowSounds features both appear on the Sound tab of the Accessibility Options dialog box shown in Figure 6-14. Both features control how Windows XP interacts with sound. The Use SoundSentry option tells Windows XP to display a visual warning when a system sound occurs. The user can choose to flash the active caption bar, flash the active window, or flash the desktop.
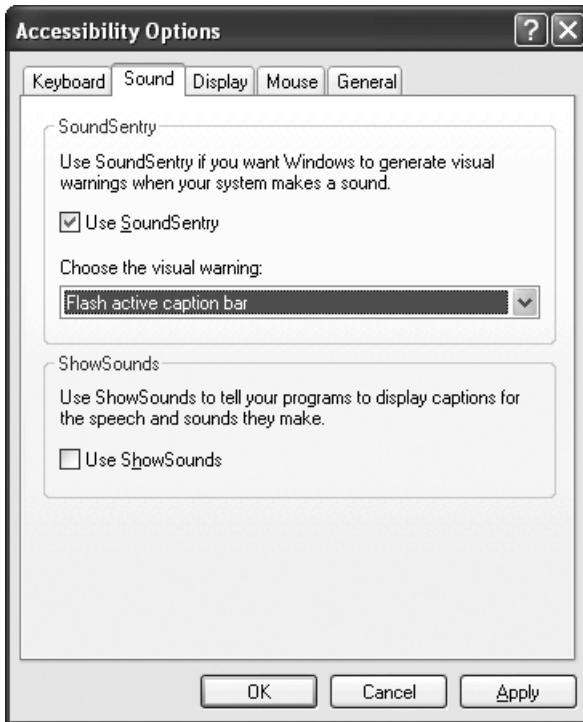
Figure 6-14. The SoundSentry and ShowSounds features determine how Windows reacts to sound events.

Unfortunately, you'll need a copy of the Tweak UI utility to control the number of times the system flashes in response to a sound. We discussed this utility in the "Flashing Text and Other Blinking Issues" section of Chapter 4, so I won't discuss it again.

Using the ShowSounds Feature

The ShowSounds feature tells Windows XP and your applications to display captions for the sounds they make. This includes speech. Instead of actually making the sound, the system requests that the application provide a description of the sound in a balloon help dialog. The only problem is that the application developer needs to intercept the request and act on it. We'll look at an example of the ShowSounds feature in action in the "Using the Sound Features Example" section of the chapter.

> **NOTE**   The use of sound presents another situation where the theory of accessibility doesn't quite match the reality. While the SoundSentry and ShowSounds features sound good in theory, the SoundSentry option works more often in practice. Most applications won't display the sounds that they make as text, even if you enable the ShowSounds feature.

Switching to a High Contrast Display

The actual purpose of the high contrast setting is to help those who have poor eyesight see the display better. However, this setting is also useful for the developer to know about. For example, the high contrast screens work well when you're tired. It also works well if you're on a plane using a laptop in bright sunlight. Sometimes a high contrast screen is even the answer for presentations. Figure 6-15 shows the Display tab of the Accessibility Options dialog box.

Figure 6-15. Define features such as the cursor blink rate using this dialog box.

To use the high contrast display, check the High Contrast option. The Settings button displays the Settings for High Contrast dialog box that you'll use to adjust the high contrast settings. In general, you don't have to use the large letter setting to gain the benefits of the high contrast display. The settings include those used for normal s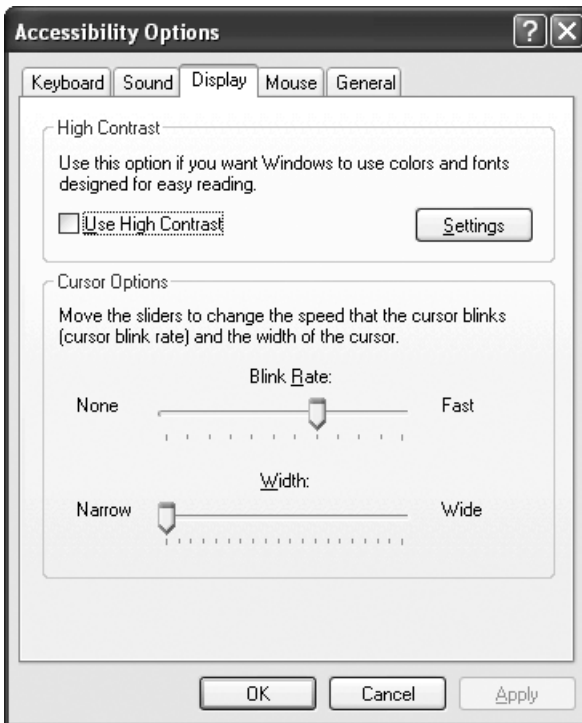ized letters as well. In addition, you can choose between a black background (good for nighttime use) or a white background (good for sunny location use).

Controlling Cursor Blink Rate and Width

At the bottom of the Display tab shown in Figure 6-15, you'll find two sliders. The Blink Rate slider controls the rate at which the cursor blinks. The Width slider controls the cursor width. Generally, these settings help someone who has problems with flashing text and other screen elements adjust their display for comfortable use. In addition, using a slower blink rate can help those with cognitive difficulties. (On the other hand, people with attention deficit disorder often need a faster blink rate in order to see the cursor.)

Like many of the other settings, you can use these settings to your advantage as well. For example, many people find that setting the cursor for a slow blink rate aids laptop use in many situations. A wider cursor can also help forms and other situations where finding the cursor might become a problem.

Accessing the MouseKeys Feature

Look at the Mouse tab of the Accessibility Options dialog box and you'll find a single option for turning MouseKeys on or off. MouseKeys enables you to use the arrow keys on the numeric keypad as a mouse. Instead of moving the cursor with the mouse, you can move it with the arrow keys. This doesn't disable your mouse; it merely augments it.

TIP MouseKeys is one of the most useful Windows Accessibility features for designers because it provides very fine control over the mouse. For example, I often use this feature when drawing block diagrams or creating the final version of a dialog box.

Click Settings on this tab to display the Settings for MouseKeys dialog box shown in Figure 6-16. Using the Pointer Speed options, you can optimize the performance of this particular feature. The Top Speed slider helps you to adjust the fastest speed at which you can move the mouse cursor using the arrow keys. The

Acceleration slider determines how quickly the cursor reaches full speed after you press it. Windows doesn't start the cursor at full speed; it brings it there gradually. The combination of these two settings determines how much added control MouseKeys gives you over the cursor. The check box in this group provides another option. You can press the Ctrl key to speed up the mouse cursor and press the Shift key to slow it down.
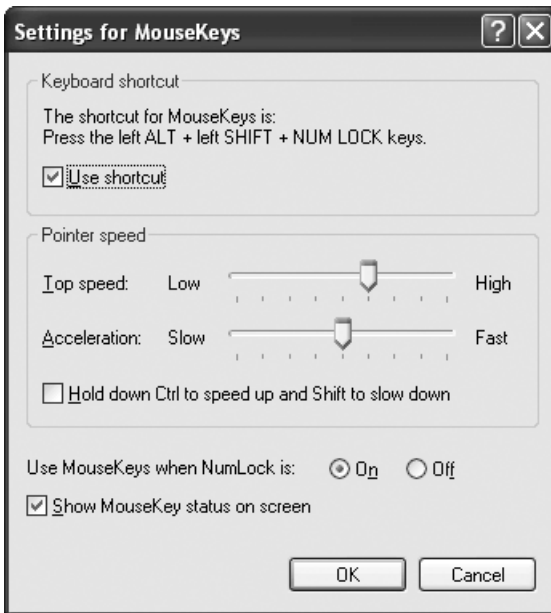
Figure 6-16. The Settings for MouseKeys dialog box enables you to change how this feature works.

There are two settings at the bottom of the dialog box. The radio buttons control when MouseKeys is active. You must specify whether the Num Lock key should be on or off when you use MouseKeys. The second option determines whether the MouseKeys icon appears on the Taskbar.

Using the Keyboard Features Example

Generally, you don't want to spend a lot of time manipulating the user environment in Windows applications. The reason is simple—if the user wants to change their environment, they have plenty of ways in which to accomplish the task. You do want to monitor the environment at all times, however, to ensure that your appli-

cation works in a way that's consistent with the current user settings (such as when the user selects a large text display).

There are times when you'll want to provide the user with configuration options—a sort of shortcut to beneficial environmental changes. You don't want the application to modify these settings, but you do want to give the user quick access to them so that they don't have to leave the application environment. In most cases, this is a convenience option that will keep the user happy and make your application easier to use, but it isn't a requirement for accessibility. For example, what happens when you've developed a new graphics application and want to be sure that the user has the ability to use accessibility features as needed? The user can always turn on the StickyKeys feature by pressing the Shift key five times, or they can open the Accessibility Options dialog box. Both of these methods are inconvenient, but they work. However, if you want to make your application truly usable, it's better if you help the user turn StickyKeys on and off as needed directly from your application. Providing a simple menu option doesn't impair the user's ability to control their environment, but it does make the application infinitely easier to use.

The example in this section shows how to create menus that will help the user control the Windows Accessibility environment. You'll want to exercise some discretion in implementing this feature, but it's important to understand that there are times when you might want to do so. For example, I often use the MouseKeys option in my drawing applications because I lack a drawing tablet and must rely on the mouse or the keyboard. Using MouseKeys makes drawing extremely fast and accurate. I wish some graphics designers would include a switch for this option in their application, but so far, none have.

The following sections describe the programming interface. You need to understand the interface because each Windows Accessibility feature has a different control. The .NET Framework doesn't provide a method to manipulate the Windows Accessibility features yet, so I had to build a library named AccesFuncs that relies on Platform Invoke (PInvoke) to perform the task. This second section gets a little technical and you don't have to know how it works in order to use it. Feel free to skip this section if desired. Finally, we'll create an application that uses the AccessFuncs library. I've divided this application into functional areas and provided some tips on how to maximum user choices. You'll need to read all of the sections to get the big picture on how Windows Accessibility works, but you can read just the section you need to implement a specific Windows Accessibility feature in your application.

Understanding the Interface

One of the first problems you'll notice with the .NET Framework is a lack of support for direct Windows Accessibility feature manipulation. You can check the status of

the HighContrast, CursorSize, and ShowSounds settings using properties in the SystemInformation class, but that's about it. We've already discussed the HighContrast option as part of the "Creating the Font Modification Example" in Chapter 5. The ShowSounds feature appears in the "Using the Sound Features Example" section of this chapter. The CursorSize feature appears in the "Using the Display Features Example" section of this chapter. We'll also discuss some mouse functionality you need to know about in the "Using the Mouse Features Example" section, but essentially, you can't even access the MouseKeys setting.

Because the .NET Framework doesn't provide the functionality needed, you'll have to use PInvoke to perform the task. Using PInvoke means learning about the Win32 API and understanding how to work with it. Obviously, the first question we need to answer is what Accessibility features Windows provides.

In general, you'll use the GetSystemMetrics() function to get the current on or off status of a standard Windows Accessibility function. You'll use the SystemParamtersInfo() function to enable these functions or to check the current user settings for them. Both of these functions rely on constants to determine which Windows Accessibility feature you want to access. Table 6-1 contains a complete list of the Windows Accessibility features, the access constant you use to access them, and a short description of how the user generally benefits from the Accessibility feature.

Table 6-1. Windows Accessibility Features

Feature	Get Status Constant	Set Status Constant	Description
StickyKeys	SPI_GETSTICKYKEYS	SPI_SETSTICKYKEYS	Makes the Ctrl, Alt, and Shift keys sticky, which means you can press them first and then the associated alphanumeric or function key. This allows users who can only press one key at a time use complex keyboard accelerators.
FilterKeys	SPI_GETFILTERKEYS	SPI_SETFILTERKEYS	Forces Windows to ignore brief or quickly repeated keystrokes. Also slows the repeat rate for the keyboard when a key is held down constantly.

Table 6-1. Windows Accessibility Features (Continued)

Feature	Get Status Constant	Set Status Constant	Description
ToggleKeys	SPI_GETTOGGLEKEYS	SPI_SETTOGGLEKEYS	Sounds a tone whenever the Caps Lock, Num Lock, Scroll Lock keys are pressed. You may need to disable this feature when an application requires extensive use of these keys, but remember to enable it again when you exit the program.
SoundSentry	SPI_GETSOUNDSENTRY	SPI_SETSOUNDSENTRY	Displays a visual warning whenever the system makes a sound. You'll want to disable non-imperative sounds when this option is used.
ShowSounds	SPI_GETSHOWSOUNDS	SPI_SETSHOWSOUNDS	Displays a caption whenever speech or sound occurs. See the "Using SAMI to Improve Your Applications" sidebar for additional details on how closed captioning can help your application.
HighContrast	SPI_GETHIGHCONTRAST	SPI_SETHIGHCONTRAST	This feature uses high contrast colors and large fonts to make reading the screen easier. If your application has formatted displays (like those for database- or dialog-based applications), you may need to adjust the display to make this feature useful.

Table 6-1. Windows Accessibility Features (Continued)

Feature	Get Status Constant	Set Status Constant	Description
MouseKeys	SPI_GETMOUSEKEYS	SPI_SETMOUSEKEYS	Allows the user to use the numeric keypad keys in place of the mouse to move the mouse cursor on screen. This feature shouldn't require changes to most applications. However, you may want to offer this feature when precise mouse placement is required, since using the cursor keys usually produces more accurate results.
SerialKey	SPI_GETSERIALKEYS	SPI_SETSERIALKEYS	Allows alternative access to mouse and keyboard features. In general, you'll never need to directly access this feature in an application (unless the application requires a special input device).

Using SAMI to Improve Your Applications

Closed captioning has moved from your television to the computer. Closed captioning is an alternate form of audio content. It may include descriptions of sounds, symbols, icons, or text to represent the audio content.

The use of closed captioning in applications affects many people. The most obvious users are persons with hearing loss or impairment. It's also useful for people who are learning to read, learning a second language, or in situations where noise is unwelcome, such as libraries or multiuse offices.

Microsoft's Synchronized Accessible Media Interchange (SAMI) improves delivery of closed captioning with multimedia applications by time-synchronizing the captioning file with the media file. This makes it easier to edit or change either file than it is in other applications where the application encodes the accessibility information within the media file.

SAMI can provide closed captioning in more than one language and with different presentation possibilities. The user chooses the appearance of the captions by selecting the color, font, and size of the text. This will increase the ease of reading for children and for people with slight visual impairments, who can select larger

screen types. The user also chooses what language the text will appear in, such as American English or Canadian French.

SAMI files use the extension .SMI or .SAMI. The file format specification is free (no licensing fee). You can find the specification, demonstrations, and examples at `http://www.microsoft.com/enable/`, which is Microsoft's Accessibility Web site. The demonstrations require Internet Explorer 4 or above (`http://www.microsoft.com/windows/ie/default.asp`) and Windows Media Player (`http://www.microsoft.com/windows/windowsmedia/download/default.asp`).

SAMI instructions look similar to HTML or XML, but the actual implementation is different. The use of a common programming idiom makes it an easy format to learn. For example, documents begin with a `<SAMI>` tag and end with a `</SAMI>` tag, replacing the `<HTML>` tag in a normal HTML document. (The tags must be uppercase.) Once you have the `<SAMI>` tag in place, a SAMI document looks much the same as its HTML counterpart. It even includes the `<HEAD>` and `<BODY>` tags. You can learn more about SAMI at `http://msdn.microsoft.com/library/en-us/dnacc/html/atg_samiarticle.asp`.

A Quick Description of the AccessFuncs Library

The AccessFuncs library encapsulates all of the code required to work with the Windows Accessibility functions. If you want, you can simply skip this section and use the library as needed in your applications. All you need is a reference to the library and the appropriate using statement in your code. The library has full documentation, so you can read the descriptions in the Object Browser to understand how the library works. In addition, the "Developing the Windows Accessibility Status Application" section describes how to use the library for application development. Of course, you'll eventually want to know how the library works so that you can modify it to meet specific needs.

As previously mentioned, this library contains all of the Win32 API access code you'll need, including data structures. Listing 6-2 provides an overview of the essential functions, enumerations, and data structures. In fact, the listing only contains one sample of each type. The extensive comments are also removed for the sake of clarity. You can see the full listing of this library in the \Chapter 06\ AccessFuncs folder of the source code that you can obtain from the Downloads section of the Apress Web site (`http://www.apress.com`).

Listing 6-2. An Overview of the AccessFuncs Library

```
public enum AccessType : uint
   {
       SPI_GETHIGHCONTRAST  = 0x0042,
       SPI_SETHIGHCONTRAST  = 0x0043,
       SPI_GETSCREENREADER  = 0x0046,
       SPI_SETSCREENREADER  = 0x0047,
       SPI_GETFILTERKEYS    = 0x0032,
       SPI_SETFILTERKEYS    = 0x0033,
       SPI_GETTOGGLEKEYS    = 0x0034,
       SPI_SETTOGGLEKEYS    = 0x0035,
       SPI_GETMOUSEKEYS     = 0x0036,
       SPI_SETMOUSEKEYS     = 0x0037,
       SPI_GETSHOWSOUNDS    = 0x0038,
       SPI_SETSHOWSOUNDS    = 0x0039,
       SPI_GETSTICKYKEYS    = 0x003A,
       SPI_SETSTICKYKEYS    = 0x003B,
       SPI_GETACCESSTIMEOUT = 0x003C,
       SPI_SETACCESSTIMEOUT = 0x003D,
       SPI_GETSERIALKEYS    = 0x003E,
       SPI_SETSERIALKEYS    = 0x003F,
       SPI_GETSOUNDSENTRY   = 0x0040,
       SPI_SETSOUNDSENTRY   = 0x0041
   }

public enum WinIniFlags
   {
       SPIF_NONE            = 0x0000,
       SPIF_UPDATEINIFILE   = 0x0001,
       SPIF_SENDWININICHANGE = 0x0002,
       SPIF_SENDCHANGE      = SPIF_SENDWININICHANGE
   }

public enum HighContrastFlags
   {
       HCF_HIGHCONTRASTON  = 0x00000001,
       HCF_AVAILABLE       = 0x00000002,
       HCF_HOTKEYACTIVE    = 0x00000004,
       HCF_CONFIRMHOTKEY   = 0x00000008,
       HCF_HOTKEYSOUND     = 0x00000010,
       HCF_INDICATOR       = 0x00000020,
       HCF_HOTKEYAVAILABLE = 0x00000040
   }
```

```
[StructLayout(LayoutKind.Sequential, Pack=1, CharSet=CharSet.Auto)]
 public struct HIGHCONTRAST
   {
      public UInt32   cbSize;
      public Int32    dwFlags;
      [MarshalAs(UnmanagedType.LPWStr, SizeConst=80)]
      public String   lpszDefaultScheme;
   }

public class Accessible
{
   public Accessible()
   {
   }

   [DllImport("User32.DLL", CharSet=CharSet.Auto, SetLastError=true)]
   public static extern bool SystemParametersInfo(AccessType uiAction,
                                                  UInt32 uiParam,
                                                  IntPtr pvParam,
                                                  WinIniFlags fWinIni);

   public const Int32 SM_SHOWSOUNDS = 70;

   [DllImport("User32.DLL", CharSet=CharSet.Auto, SetLastError=true)]
   public static extern Int32 GetSystemMetrics(Int32 nIndex);
}
```

The actual place to start in this listing is the `Accessible` class. This class con-
tains two functions. The `GetSystemMetrics()` function only performs one task—it
obtains the current ShowSounds state. You call it using the `SM_SHOWSOUNDS` constant.
The `GetSystemMetrics()` function returns a value indicating whether ShowSounds is
on or off. You can replicate this functionality using the `SystemInformation.ShowSounds`
property, so generally you should avoid using this function. It's provided in
the interest of completeness and for those times when you want to verify the
ShowSounds status.

The `SystemParametersInfo()` function isn't replicated anywhere within the .NET
Framework, so you'll find use for this function in your toolkit. Notice that you feed
the function four arguments. The first is one of the members of the AccessType enu-
meration that appears at the beginning of the listing. Notice that there's a get and set
member for each of the accessibility functions. Consequently, if you want to obtain
the current HighContrast status, you use the `SPI_GETHIGHCONTRAST` value and if you
want to set the HighContrast feature, you use the `SPI_SETHIGHCONTRAST` value.

The second and third arguments are related. The `uiParam` argument contains the size of the data structure passed as the `pvParam` argument. We'll see how you obtain this information in the "Developing the Windows Accessibility Status Application" section of the chapter. For now, you need to know that the data structure for each accessibility function is different. The `HIGHCONTRAST` data structure shown in the listing is representative. Every one of the data structures will contain the `cbSize` and the `dwFlags` members shown. The `cbSize` member contains the size of the data structure. The `dwFlags` member contains a numeric value that you interpret using the bit positions in the associated flag enumeration (`HighContrastFlags` in this case). Most of the data structures also contain some type of specialized information. For example, the `HIGHCONTRAST` data structure contains the name of the default high contrast scheme. Notice the technique used to marshal the string from the unmanaged environment. You must tell the Common Language Runtime (CLR) what type of string to create and how large to make it; otherwise, the function call will fail because the Win32 API will lack essential information.

The fourth argument is one of the members of the `WinIniFlags` enumeration. This value determines how Windows updates the user's profile. In general, you don't want to change the user's profile without the user's permission, so you'll usually set this entry to `SPIF_NONE`.

When the application makes a call to the `SystemParametersInfo()` function, the system will return a Boolean value indicating if the call is successful. Notice the `SetLastError=true` argument in the `[DllImport]` attribute. This argument tells the CLR to save any error information it receives. If the application detects an error in the function call, it should use the `Marshal.GetLastWin32Error()` method to retrieve the error number. Never use the Win32 API `GetLastError()` function to retrieve this information because this function could return unreliable or incorrect results from within the .NET environment.

Developing the Windows Accessibility Status Application

At this point, we have a library that can get and set the various Windows Accessibility feature values. In many cases, you'll never touch the settings, but will need the current setting values so that you can create a truly accessible application. The purpose of this section is to provide a quick overview of how you'd use the AccessFuncs library in an application. Of course, the first task you'll perform is to add a reference to the library to your application.

The sample application uses a standard menu to display the current Windows Accessibility feature status and help the user switch the feature on or off. You could easily add such a menu to your application or make it part of an options dialog. The point is that adding this functionality to your application as needed makes the Windows Accessibility feature easier to use.

Listing 6-3 provides an overview of the code for this example. You'll find a complete listing for this application in the \Chapter 06\AccessSettings folder of the source code that you can obtain from the Downloads section of the Apress Web site (http://www.apress.com).

Listing 6-3. An Overview of the AccesFuncs Library Test Application

```
public frmMain()
{
   Int32            DataSize;   // Size of the data structure.

   // Required for Windows Form Designer support
   InitializeComponent();

   // Initialize the data structures.
   AT = new ACCESSTIMEOUT();    // Access Timeout
   FK = new FILTERKEYS();        // FilterKeys
   HC = new HIGHCONTRAST();      // High Contrast
   MK = new MOUSEKEYS();         // MouseKeys
   SR = false;                   // ScreenReader
   SC = new SERIALKEYS();        // SerialKeys
   SSound = false;               // ShowSounds
   SSentry = new SOUNDSENTRY();  // SoundSentry
   SK = new STICKYKEYS();        // StickyKeys
   TK = new TOGGLEKEYS();        // ToggleKeys

   // Initialize the data structures and Choose menu options. The
   // process includes getting the current option status (which
   // fills out the data structure) and then comparing the flag
   // values to see if the option is on.

   // … Some example code left out here…

   // High Contrast
   DataSize = Marshal.SizeOf(HC);
   HC.cbSize = Convert.ToUInt32(DataSize);
   HC = (HIGHCONTRAST)GetAccessibleOption(
                    HC,
                    DataSize,
                    AccessType.SPI_GETHIGHCONTRAST,
                    WinIniFlags.SPIF_NONE);
   if ((HC.dwFlags & (Int32)HighContrastFlags.HCF_HIGHCONTRASTON) ==
                    (Int32)HighContrastFlags.HCF_HIGHCONTRASTON)
      mnuChooseHighContrast.Checked = true;
```

```
   // Screen Reader
   DataSize = Marshal.SizeOf(SR);
   SR = (bool)GetAccessibleOption(
             SR,
             DataSize,
             AccessType.SPI_GETSCREENREADER,
             WinIniFlags.SPIF_NONE);
   if (SR)
      mnuChooseScreenReader.Checked = true;


   // SerialKeys
   DataSize = Marshal.SizeOf(SC);
   SC.cbSize = Convert.ToUInt32(DataSize);
   SC = (SERIALKEYS)GetAccessibleOption(
                   SC,
                   DataSize,
                   AccessType.SPI_GETSERIALKEYS,
                   WinIniFlags.SPIF_NONE);
   if ((SC.dwFlags & (Int32)SerialKeysFlags.SERKF_SERIALKEYSON) ==
      (Int32)SerialKeysFlags.SERKF_SERIALKEYSON)
      mnuChooseSerialKeys.Checked = true;
   else if (SC.lpszActivePort == null)

      // This is one of the few accessibility options not supported
      // under Windows 2000/XP. Microsoft changed this behavior to
      // ensure that SerialKeys devices would appear as standard
      // input devices to the application.  The lpszActivePort member
      // will always contain a value for operating systems that
      // support the SerialKeys feature.
      mnuChooseSerialKeys.Enabled = false;
// … Some example code left out here…
}

private Object GetAccessibleOption(Object Struct,
                                   Int32 StructSize,
                                   AccessType AccessType,
                                   WinIniFlags IniFlag)
{
   Object   ReturnValue;   // The return data.

   // Allocate enough memory to create an unmanaged version
   // of the data structure.
   IntPtr   DataPtr = Marshal.AllocHGlobal(StructSize);
```

```
         // Point to the managed data stucture using the unmanaged
         // memory pointer.
         Marshal.StructureToPtr(Struct, DataPtr, true);

         // Call the SystemParametersInfo() function using the
         // unmanaged data structure pointer.
         Accessible.SystemParametersInfo(AccessType,
                                         Convert.ToUInt32(StructSize),
                                         DataPtr,
                                         IniFlag);

         // Move the data retrieved from the unmanaged environment to
         // the managed data structure and return this data structure
         // as an object.
         ReturnValue = Marshal.PtrToStructure(DataPtr, Struct.GetType());

         // Deallocate the memory we previously allocated.
         Marshal.FreeHGlobal(DataPtr);

         // Return the data.
         return ReturnValue;
      }

      private bool SetAccessibleOption(Object Struct,
                                       Int32 StructSize,
                                       AccessType AccessType,
                                       WinIniFlags IniFlag)
      {
         bool  ReturnValue;   // The return value of this method.

         // Allocate enough memory to create an unmanaged version
         // of the data structure.
         IntPtr   DataPtr = Marshal.AllocHGlobal(StructSize);

         // Point to the managed data stucture using the unmanaged
         // memory pointer.
         Marshal.StructureToPtr(Struct, DataPtr, true);

         // Return true if the SystemParametersInfo() function call
         // successfully modifies the Windows Accessibility features
         // using the data in the data structure.
```

```
   ReturnValue = Accessible.SystemParametersInfo(
      AccessType,
      Convert.ToUInt32(StructSize),
      DataPtr,
      IniFlag);

   // Deallocate the memory we previously allocated.
   Marshal.FreeHGlobal(DataPtr);

   // Return the data.
   return ReturnValue;
}


private void mnuChooseHighContrast_Click(object sender, System.EventArgs e)
{
   Int32 DataSize;    // Size of the data structure.

   // Set the flag value as needed to toggle the feature on or off.
   if ((HC.dwFlags & (Int32)HighContrastFlags.HCF_HIGHCONTRASTON) ==
                     (Int32)HighContrastFlags.HCF_HIGHCONTRASTON)
      HC.dwFlags = HC.dwFlags ^
          (Int32)HighContrastFlags.HCF_HIGHCONTRASTON;
   else
      HC.dwFlags = HC.dwFlags |
          (Int32)HighContrastFlags.HCF_HIGHCONTRASTON;

   // Call on the library function to set the new FilterKeys status.
   DataSize = Marshal.SizeOf(HC);

   // If the function fails, display an error message.
   if (!SetAccessibleOption(HC,
                            DataSize,
                            AccessType.SPI_SETHIGHCONTRAST,
                            WinIniFlags.SPIF_NONE))
      MessageBox.Show("Could not set the High Contrast option",
                     "Accessibility Option Error",
                     MessageBoxButtons.OK,
                     MessageBoxIcon.Error);

   // If the function succeeds, display a success message and change
   // the menu setting.
   else
      mnuChooseHighContrast.Checked = !mnuChooseHighContrast.Checked;
}
```

The application begins by obtaining the current Windows Accessibility feature status in the constructor. In a production application, you'd probably want to check this status every time the user makes some type of request. In this case, that would mean getting the status every time the user opens the Choose Feature menu. The reason for this constant vigilance is that someone could modify the Windows Accessibility feature settings from outside of your application. Although the listing only shows a few of the feature requests, the complete application requests the status of all Windows Accessibility features.

Obtaining the HighContrast feature status is representative of most of the Windows Accessibility features. The application uses the `Marshal.SizeOf()` method to obtain the size of the data structure. It uses this information to fill in the `cbSize` data member and also as input to the `GetAccessibleOption()` method that we'll discuss later in this section. The `GetAccessibleOption()` method returns an `Object` data type since the same method is used for all of the Windows Accessibility feature calls. This means you have to convert the output to the correct data type. The code checks the `dwFlags` data member for the `HCF_HIGHCONTRASTON` setting. If this flag bit is set, then the code checks the HighContrast menu entry.

The Screen Reader is one of two Windows Accessibility features that doesn't require a data structure (ShowSounds is the other). The technique used in this case differs from a Windows Accessibility feature that requires a data structure, but the principle is the same. Notice that we still have to obtain the size of the variable and pass it along with the other information to the `GetAccessibleOption()` method. The output is a `bool`, so the code can look at this value directly.

The SerialKeys Windows Accessibility feature represents a special case. Older versions of Windows provide access to this feature, so you can read the various settings and provide accommodation for the devices attached using SerialKeys in your code. Newer versions of Windows don't provide this access using the theory that a SerialKeys device shouldn't look any different than any other device attached to the system. In sum, the device shouldn't require any special handling. In some ways, this viewpoint is justified, but it would still be nice to be able to turn the device on or off as needed.

The `GetAccessibleOption()` method comes next. The first task this method performs is to allocate memory using the `Marshal.AllocHGlobal()` method. At this point, you might wonder why we have to perform this task. Remember that your .NET application uses managed memory that the Garbage Collector controls. This call is to the unmanaged environment, where we need to use unmanaged memory. The `Marshal.AllocHGlobal()` method allocates unmanaged memory for this purpose.

The code calls the `Accessible.SystemParametersInfo()` method next. I described this method in the "A Quick Description of the AccessFuncs Library" section of the chapter. On return from this call, `DataPtr` (the unmanaged memory) contains the data we need. The code uses the `Marshal.PtrToStructure()` method to move the data from unmanaged memory into managed memory.

The next step is to free the unmanaged memory because we don't need it anymore. The code uses the `Marshal.FreeHGlobal()` method to perform this task. The `GetAccessibleOption()` method ends by returning the object containing the data to the caller. As mentioned earlier, the caller still needs to convert this generic object into a specific data structure to read the values.

You'll immediately notice that the `SetAccessibleOption()` method is similar, but not precisely the same as the `GetAccessibleOption()` method. The code still allocates and deallocates memory manually. In addition, it still relies on the `Accessible.SystemParametersInfo()` method to set the Windows Accessibility information. Because we're not getting new information, in this case, the code can simply return a `bool` value indicating success or failure.

The `mnuChooseHighContrast_Click()` method is representative of the event handlers for the menu. When a user clicks one of the menu entries, the code has to modify the flag values of the appropriate data structure to turn the feature on or off. The code shows one technique for performing this task. Once the data structure is modified, the code determines the size of the data structure and calls on the `SetAccessibleOption()` method to set the new value. The final step is to check or clear the menu option so that the user can see the current Windows Accessibility option status.

As you go through the code on the disk, you'll notice that some of the event handlers simply display a message, rather than change an option. For example, you can't turn on the screen reader from the application. To perform this task, you must start the associated application. In sum, you can't modify some Windows Accessibility features with this application because it doesn't make sense to do so. However, you can always retrieve the status information, which is a lot more than the .NET Framework allows you to do. Look through the various data structures to determine what types of information are available.

Using the Sound Features Example

Many .NET developers have already run across one problem with sound in this environment—there isn't any support built in for it unless you're using Visual Basic and you're happy with a plain beep. Most developers want something better than a plain beep, which means resorting to PInvoke in .NET. Using the Win32 API `PlaySound()` function enables the developer to play both standard system sounds as well as other media types such as WAV files.

Given the purpose of this book, you might wonder why I'm worried about sound. It turns out that there's also little support for the ShowSounds Windows Accessibility feature in most Windows applications today. In fact, finding an application that supports this feature is difficult at best.

This example starts with the premise that combining these two needs into one component would prove very convenient. The PlaySound control plays a sound and works with ShowSounds at the same time. All you need to do to use it is define a few properties. The sections that follow tell how this control works and demonstrates its use within a simple application.

Creating the PlaySound Control

The PlaySound control performs two tasks. First, it plays a sound. Second, it displays a tooltip if the user turns the ShowSounds feature on. The two tasks are independent of each other. The control can play a sound independently of the ShowSounds feature setting, so it's possible for the control to perform both or either task.

TIP This control opens some interesting usage possibilities. For example, a developer could set an application up so that it simply displayed sound descriptions for users in office environments. The sound description would still notify the user of an event without disturbing the user's neighbors.

Now that you have some idea of what this control will do, let's look at the code. Listing 6-4 contains a partial listing of the code for this example. You'll find a complete a complete listing in the \Chapter 06\PlaySound folder of the source code that you can obtain from the Downloads section of the Apress Web site (http:// www.apress.com).

Listing 6-4. Providing Sound and ShowSounds Functionality with the PlaySound Control

```
public class PlaySound : Component
{
   public PlaySound()
   {
      // Initialize the property values.
      _MakeSound = true;
      _SoundFileName = "";
      _ShowSoundsDescription = "The System Plays a Sound";
      _AutomaticDelay = 300;
      _AutoPopDelay = 7000;
      NoShow = new Timer();
   }
```

```
/// <summary>
/// This event fires when the system displays a
/// ShowSounds message.
/// </summary>
public event EventHandler SoundDisplayed;


/// <summary>
/// This event fires when the system generates a
/// sound.
/// </summary>
public event EventHandler SoundGenerated;


/// <summary>
/// Determines if the system will make a sound.
/// </summary>
public bool MakeSound
{
   get {return _MakeSound;}
   set {_MakeSound = value;}
}


// This property requires a special editor to ensure it works
// as intended.
/// <summary>
/// Contains the name of the file to play.
/// </summary>
[EditorAttribute(typeof(FileNameEditor), typeof(UITypeEditor))]
public string SoundFileName
{
   get
   {
      return _SoundFileName;
   }
   set
   {
      _SoundFileName = value;
   }
}
```

```csharp
/// <summary>
/// Describes the sound to the person using ShowSounds.
/// </summary>
public string ShowSoundsDescription
{
   get {return _ShowSoundsDescription;}
   set
   {
      if (value != null)
         _ShowSoundsDescription = value;
   }
}


/// <summary>
/// Determines the hover delay for the tooltip
/// displaying the sound description. (1 ms minimum)
/// </summary>
public int AutomaticDelay
{
   get {return _AutomaticDelay;}
   set
   {
      if (value >= 1)
         _AutomaticDelay = value;
   }
}


/// <summary>
/// Determines the amount of time the tooltip
/// will appear on screen. (3000 ms minimum)
/// </summary>
public int AutoPopDelay
{
   get {return _AutoPopDelay;}
   set
   {
      if (value >= 3000)
         _AutoPopDelay = value;
   }
}
```

```
/// <summary>
/// Outputs a sound and/or a sound description based
/// upon the current ShowSound Windows Accessiblity setting.
/// </summary>
/// <param name="Parent">The control hosting the sound.</param>
public void GenerateSound(Control Parent)
{
   System.EventArgs  EA;    // Used When Raising an Event.

   // Initialize the EventArgs.
   EA = new EventArgs();

   // Initialize the ToolTip.
   TT = new ToolTip();
   TT.AutomaticDelay = _AutomaticDelay;
   TT.AutoPopDelay = _AutoPopDelay;

   // Intitialze the Timer.
   NoShow.Interval = _AutoPopDelay;
   NoShow.Tick += new EventHandler(this.NoShow_Tick);

   // If the ShowSounds option is selected, display text.
   if (IsShowSoundsSelected())
   {
      // Display the information on screen.
      TT.SetToolTip(Parent, _ShowSoundsDescription);
      TT.Active = true;
      NoShow.Start();

      // Fire the event.
      if (SoundDisplayed != null)
         SoundDisplayed(this, EA);
   }

   if (_MakeSound)
   {
      // Play a sound only when the user requests it.
      WinPlaySound(@_SoundFileName,
         0,
         SND_FILENAME | SND_ASYNC);
```

```
            // Fire the event.
            if (SoundGenerated != null)
                SoundGenerated(this, EA);
        }
    }


    /// <summary>
    /// Obtains the current status of the ShowSounds
    /// Windows Accessibility setting.
    /// </summary>
    /// <returns>True or False depending on setting value</returns>
    public bool IsShowSoundsSelected()
    {
        return SystemInformation.ShowSounds;
    }


    // Define some constants for using the PlaySound() function.
    private const int SND_SYNC = 0x0000;
    private const int SND_ASYNC = 0x0001;
    private const int SND_FILENAME = 0x00020000;


    // Import the Windows PlaySound() function.
    [DllImport("winmm.dll",
        EntryPoint="PlaySound", CharSet=CharSet.Auto, SetLastError=true)]
    private static extern bool WinPlaySound(string pszSound,
                                            int hmod,
                                            int fdwSound);


    private void NoShow_Tick(Object sender, System.EventArgs e)
    {
        // Stop displaying the sound description.
        TT.Active = false;

        // Stop the timer.
        NoShow.Stop();
    }
}
```

As you can see, it's a lot of code (and I cut it down for the purposes of display in the book). The first thing you should notice is that I've based this control on the Component class. The reason for this choice is that the control doesn't include a window of any sort—it's more akin to a timer than to a command button.

The constructor begins by initializing all of the property values for the controls. Notice that we use private variables for the properties. These values are exposed by the properties that appear later in the listing.

The control supports two events. The control fires these events whenever it displays a sound description or generates a sound. Generally, you won't need to handle the events, but sometimes an application needs to know when these events actually occur.

All of the properties come next in the listing. Most of the properties are of the generic get/set variety. Notice that the `SoundFileName` property includes support for an editor. This editor displays a File Open dialog box that helps the control user locate the sound file on disk. This control only supports external sound files, but you could easily extend it to support embedded resources and system sounds.

I wanted to ensure that the control will always have some type of value to display as a sound description, so the `ShowSoundsDescription` property looks for a `null` value. If the value isn't `null`, it will set the sound description to that value. A production version of this control would probably include additional checks to ensure that the developer provided something that at least looks like a sound description. Of course, there are limits to what the control can check, so someone who really doesn't want to display a usable sound description is certainly free to come up with something less than usable.

The `AutomaticDelay` and `AutoPopDelay` properties set timer values for the control, so it's important to ensure that the developer provide realistic values. Both properties look for numeric input that's greater than a baseline value. If the input doesn't meet this requirement, the control uses the current value instead.

The main method for this control is `GenerateSound()`. This method displays sound descriptions on screen and generates the audible sound as needed. The code begins by creating and initializing the variables used within the control. For example, this is where the `ToolTip`, `TT`, is initialized.

The first check the code makes is to verify the state of ShowSounds using the `IsShowSoundsSelected()` method. The `IsShowSoundsSelected()` method simply returns the state of the `SystemInformation.ShowSounds` property. The reason the control uses this technique is to expose this functionality to the developer as well. This makes the `SystemInformation.ShowSounds` property easier for the developer using the control to access. Otherwise, the `GenerateSound()` method could have accessed the `SystemInformation.ShowSounds` property directly.

If ShowSounds is active, the code associates `TT` with the parent control passed into the `GenerateSound()` method by the caller. It displays `TT` so the user can see the description associated with the sound. Unfortunately, `TT` doesn't go away if you leave it in this state. You have to deactivate it. In this case, we'll use a timer, `NoShow`, to perform that task. So, the next step the code performs is to start `NoShow`, which has already been set up with the `_AutoPopDelay` value. The final step is to fire the `SoundDisplayed` event so that the application knows the event occurred.

The next task that `GenerateSound()` performs is to check the `_MakeSound` value. If the developer chooses to make a sound, the code calls `WinPlaySound()`, which calls a Win32 API function to generate the audible sound. The final step is to fire the `SoundGenerated` event.

As you can see from the listing, the `WinPlaySound()` function is simply a declaration of an imported Win32 API function, `PlaySound()`. In this case, the declaration requires an `EntryPoint` argument to ensure that the `[DllImport]` attribute can locate the proper call. The `PlaySound()` function accepts three inputs, only two of which we need to provide in this case. The first input is a string that describes the name and location of the sound. In this case, the sound is always a filename. However, you can also specify an internal application resource, the name of a system sound in the registry, or the name of a sound in the WIN.INI file. We don't use the `hmod` argument in this case, but you must use it when you want to use an application resource as input. Finally, the `fdwSound` argument contains flags that determine how Windows plays the sound. For example, you can select between asynchronous (where control is returned immediately) and synchronous sound playing.

The `NoShow_Tick()` event handler is the last essential piece of the control. A single tick is the duration that the developer wants to display the `ToolTip`, `TT`. When this single tick occurs, it's time to hide `TT` until the application wants to play a sound again. This event handler sets `TT.Active` to false so the `ToolTip` won't display when the user hovers the mouse of the control associated with the sound. The code then stops the timer so that the control is no longer active and the application can garbage collect it if desired.

Creating the ShowSounds Test Application

The ShowSounds test application is a simple test of the PlaySound control. It includes a few check boxes that enable the developer to test the events and to turn the sound-playing feature on or off. (If you want to turn ShowSounds on of off, use the Accessibility Options applet in the Control Panel.) Listing 6-5 shows the Test button and PlaySound control event handlers. You'll find the complete listing for this example in the \Chapter 06\ShowSounds folder of the source code that you can obtain from the Downloads section of the Apress Web site (http://www.apress.com).

Listing 6-5. The ShowSounds Test Application

```
private void btnTest_Click(object sender, System.EventArgs e)
{
   // Check to see if the user wants to play a sound.
   if (cbSound.Checked)
      MySound.MakeSound = true;
   else
      MySound.MakeSound = false;
```

```
   // Check to see if the user wants the SoundDisplayed event.
   if (cbSoundDisplay.Checked)
      MySound.SoundDisplayed +=
         new EventHandler(playSound1_SoundDisplayed);
   else
      MySound.SoundDisplayed -=
         new EventHandler(playSound1_SoundDisplayed);

   // Check to see if the user wants the SoundGenerated event.
   if (cbSoundGenerate.Checked)
      MySound.SoundGenerated +=
         new EventHandler(playSound1_SoundGenerated);
   else
      MySound.SoundGenerated -=
         new EventHandler(playSound1_SoundGenerated);

   // Perform the sound related task.
   MySound.GenerateSound(btnTest);
}

private void playSound1_SoundDisplayed(object sender, System.EventArgs e)
{
   // Display a message box.
   MessageBox.Show("Sound Displayed");
}

private void playSound1_SoundGenerated(object sender, System.EventArgs e)
{
   // Display a message box.
   MessageBox.Show("Sound Generated");
}
```

The btnTest_Click() method begins by checking the state of cbSound. If the user checks this control, then the application will play the sound associated with the MySound control. Likewise, the code checks the state of the cbSoundDisplay and cbSoundGenerate controls to determine if they're checked. If so, the code assigns an event handler to the affected events. In this case, the event handlers display a message box. Finally, the method calls MySound.GenerateSound() with the test button as input. Figure 6-17 shows typical output for this application.
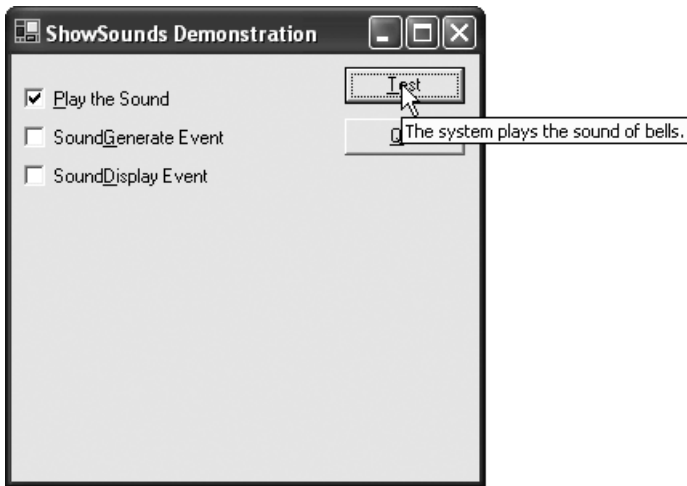
Figure 6-17. The test application helps you see the functionality of the PlaySound control.

Using the Display Features Example

We've already discussed several display-related issues, such as the use of high contrast in the book. However, we haven't discussed one important element—the cursor used to display information on screen. Checking the size of the cursor can help you format the display for easier viewing by those who want to use a large cursor size. Not only that, but checking the cursor size can often provide cues about a user's overall viewing needs. Some users don't use the HighContrast setting, even when they use a large text display to see information on screen. Listing 6-6 shows the simple technique used to determine the current cursor size. You'll find a complete listing for this example in the \Chapter 06\CursorData folder of the source code that you can obtain from the Downloads section of the Apress Web site (http://www.apress.com).

Listing 6-6. A Technique for Obtaining the Cursor Size

```
private void btnGetCursor_Click(object sender, System.EventArgs e)
{
   StringBuilder  CursorData; // Cursor information.

   // Initialize the StringBuilder
   CursorData = new StringBuilder();
```

```
  // Get the cursor information.
  CursorData.Append("The current cursor size is:\r\n\r\nHeight: ");
  CursorData.Append(SystemInformation.CursorSize.Height.ToString());
  CursorData.Append("\r\nWidth: ");
  CursorData.Append(SystemInformation.CursorSize.Width.ToString());

  // Display the information.
  MessageBox.Show(CursorData.ToString(),
                  "Cursor Data",
                  MessageBoxButtons.OK,
                  MessageBoxIcon.Information);
}
```

The centerpiece of this code is the `SystemInformation.CursorSize` property. This
property provides the information needed to find the actual height and width of the
cursor. In general, you'll find that most standard displays use a 32-pixel×32-pixel
cursor size. However, some displays use larger or smaller sizes depending on the
needs of the user and the current system configuration. The application generates
a message box containing the cursor size information.

NOTE You'll notice that most of the examples that rely on the
`SystemInformation` class in this chapter don't modify the associated
property. In general, the `SystemInformation` class only allows the devel-
oper to view the system setting—not change it. If you want to change
a system setting, you'll need to use a Win32 API function such as we
used in the "A Quick Description of the AccessFuncs Library" section
of the chapter.

Using the Mouse Features Example

One area in which the .NET Framework provides complete information to the
developer is the mouse. You still can't modify any of the information without
resorting to a Win32 API call, but at least you can determine the mouse status and
many of the common features it provides. For example, you can determine how
many buttons a mouse has and whether it supports a mouse wheel. Listing 6-7
shows the code for this example. You'll find a complete listing for this example in
the \Chapter 06\MouseCheck folder of the source code that you can obtain from
the Downloads section of the Apress Web site (`http://www.apress.com`).

Listing 6-7. Techniques for Determining the Mouse Status and Configuration

```
private void btnTest_Click(object sender, System.EventArgs e)
{
   StringBuilder  MouseData;  // The mouse setup information.

   // Initialize the StringBuilder.
   MouseData = new StringBuilder();

   // Check for the presence of a mouse.
   if (SystemInformation.MousePresent)
   {
      // Begin building the MouseData string.
      MouseData.Append("System includes a mouse with the " +
                       "following characteristics:\r\n");

      // Get the clicking information.
      MouseData.Append("\r\nDouble Click Area (pixels): ");
      MouseData.Append(
         SystemInformation.DoubleClickSize.ToString());
      MouseData.Append("\r\nDouble Click Time (ms): ");
      MouseData.Append(
         SystemInformation.DoubleClickTime.ToString());

      // Get the mouse specific information.
      MouseData.Append("\r\n\r\nNumber of Mouse Buttons: ");
      MouseData.Append(
         SystemInformation.MouseButtons.ToString());
      MouseData.Append("\r\nAre the buttons swapped? ");
      MouseData.Append(
         SystemInformation.MouseButtonsSwapped.ToString());
      MouseData.Append("\r\n\r\nIs a mouse wheel available? ");
      MouseData.Append(
         SystemInformation.MouseWheelPresent.ToString());

      // Display this information only for systems with
      // mouse wheel support.
      if (SystemInformation.MouseWheelPresent)
      {
         MouseData.Append("\r\nMouse wheel scroll lines: ");
         MouseData.Append(
            SystemInformation.MouseWheelScrollLines.ToString());
         MouseData.Append("\r\nNative mouse wheel support? ");
         MouseData.Append(
            SystemInformation.NativeMouseWheelSupport.ToString());
      }
```

```
    // Display the results.
    MessageBox.Show(MouseData.ToString(),
                    "Mouse Access Information",
                    MessageBoxButtons.OK,
                    MessageBoxIcon.Information);
}

// If no mouse is present, display a message and exit.
else
    MessageBox.Show("There is no mouse connected to this system.",
                    "No Mouse Access",
                    MessageBoxButtons.OK,
                    MessageBoxIcon.Exclamation);
}
```

As you can see from the code, all of the mouse statistics appear in the SystemInformation class. Various properties in this class tell you different things about the mouse, such as support for a mouse wheel and the current size of the double click area. Information such as the size of the double click area can cue you about the abilities of the user in some situations. A large DoubleClickSize value could indicate that the user has mobility problems—a large DoubleClickTime value tends to enforce this idea. Figure 6-18 shows the output of this example.
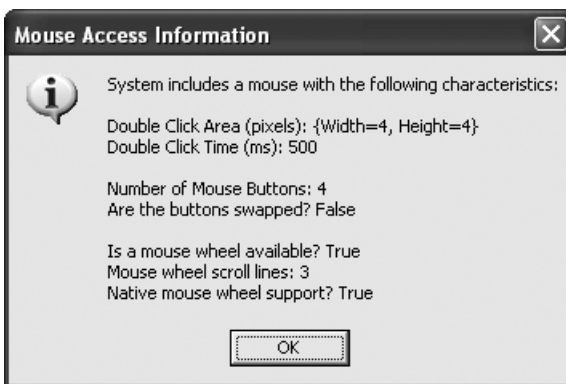

Figure 6-18. Use this test application to learn more about the functionality of the mouse attached to a machine.

Notice that the information in Figure 6-18 is complete, but also generic. For example, the target mouse provides support for four buttons. Although you know the tasks the first two buttons perform, you don't know what the other two do?

Are these buttons available for special use in your application? You won't know unless you ask the user. In short, these properties provide access to some level of automation, but perhaps not the complete automation that many developers would like.

Summary

This chapter has helped you discover the Windows Accessibility features. The main idea behind this chapter is to show that you can provide good accessibility support in an application with only a little extra coding in many cases. The amazing thing is that Microsoft built this support into Windows—the user doesn't have to spend one penny extra to get it. All that they need is for your application to recognize the added support and provide the additional code needed to activate it. Interestingly, the .NET Framework makes this process easier because Microsoft has added some functionality that you would have had to code by hand in the past.

Even so, things aren't perfect with the Windows Accessibility support. So, one of the first things you need to do is determine how far the Accessibility features go in meeting your potential user's needs. In some cases, the Accessibility features provide everything you need; but, in other cases, users will need to buy some special hardware to make the access perfect. The point is that you need to plan for this support during the design phase, ensure it actually works during the testing phase, and then make users aware of the accessibility features and requirements for your application. Otherwise, users might have unrealistic expectations that your application won't satisfy.

It's also time to work with the accessibility features by yourself. Make sure you try your application out using the accessibility features. Sometimes, the best way to learn how the accessibility features work is to do something like try them while blindfolded. Ask yourself how easy your application is to use when you can't see the screen. In this case, not seeing is believing (and learning). You'll also want to try the JAWS demonstration software mentioned in the "Using the Narrator" section of the chapter. This product demonstrates the superior performance and usability of some of the products on the market when compared to the Windows offerings.

Chapter 7 looks at usage cues. Many of the accessibility features Windows provides make application access easier, but they don't necessarily make the application easier to use. Earlier in the book, we had discussed the need for an accessibility friendly application to provide both access and usability. While this chapter concentrated on access, Chapter 7 will help you learn about usability in the world of Windows programming.