10

Detecting Devices

This chapter shows how applications can obtain information about attached devices, request a handle for communicating with a device, and detect when a device is attached or removed. Each of these tasks involve using Windows API functions and the device interface GUIDs introduced in Chapter 8. As an aid to those with limited experience with API functions, the chapter begins with a short tutorial on the topic.

A Brief Guide to Calling API Functions

You can do a lot of programming without ever calling an API function. Microsoft's .NET Framework provides classes that support common tasks such as creating user interfaces, accessing files, manipulating text and graphics, accessing common peripheral types, networking, security functions, and exception handling. Internally, the classes' methods are likely to call API functions, but the classes offer a safer, more secure, and more modular, object-oriented way to accomplish the tasks. Languages that support using

the .NET Framework include Visual Basic .NET, Visual C# .NET, and Visual C++ .NET.

But the .NET Framework doesn't handle every task. Applications still need to call API functions for some things. A .NET application can use the .NET Framework where possible and API calls when needed. Applications in languages that predate the .NET platform, such as Visual Basic 6, sometimes need to call API functions as well. The examples in this chapter are written for Visual Basic .NET and Visual C++. NET.

Because calling API functions can be an obscure art at times, this section includes an introduction to some things that are useful to know when using the Windows API.

Managed and Unmanaged Code

Understanding how to call API functions in Visual Basic .NET applications requires understanding the difference between managed and unmanaged code. Windows API functions use unmanaged code. Their DLLs contain compiled machine code that executes directly on the target CPU. In Visual Basic .NET and Visual C# .NET, all program code is managed code that compiles to the Microsoft Intermediate Language (MSIL). The .NET platform's common language runtime (CLR) environment executes the MSIL code.

Managed code has advantages. Because all .NET languages use the same CLR, components written in different .NET languages can easily interoperate. For example, a Visual Basic .NET application can call a function written in Visual C# .NET without worrying about differences in calling conventions. The CLR also simplifies programming by using garbage collection to manage memory.

A .NET application can call functions that use unmanaged code, including Windows API functions. But Visual Basic .NET and Visual C# .NET applications must take special care to ensure that any data being passed survives the trip from managed to unmanaged code, and back if necessary.

To use data returned by an API function, a Visual Basic .NET or Visual C# .NET application often must *marshal* the data to make it available to the managed code. Marshaling means doing whatever is needed to make the data available and typically involves copying data into managed memory and/or converting data from one type or format to another.

The .NET Framework's Marshal class provides methods for allocating memory that unmanaged code will use, for copying blocks of unmanaged memory to managed memory, and for converting between managed and unmanaged data types. For example, the PtrToStringAuto method accepts a pointer to a string in unmanaged memory and returns the string being pointed to. This Visual-Basic code retrieves a string from a pointer (IntPtr pdevicePathName) returned by an API function:

```
Dim DevicePathName as String
DevicePathName = _
  Marshal.PtrToStringAuto(pdevicePathName)
```

Arrays that will contain data copied from unmanaged code must use the MarshalAs attribute to define the size of the array. This Visual-Basic code declares a 16-byte array that will hold a GUID copied from a structure returned by an API call:

```
<MarshalAs(UnmanagedType.ByValArray, _
  ArraySubType:=UnmanagedType.U1, SizeConst:=16)> _
  Public dbcc_classguid() _
  As Byte
```

The GUID is marshaled into the byte array as an UnmanagedType.ByValArray. The ArraySubType field defines the array's elements as unsigned, 1-byte (U1) values and the SizeConst field sets the array's size as 16 bytes.

What about Visual C++ .NET? A Visual C++ .NET application can compile to managed code, unmanaged code, or even some of each. The language also incorporates the "It Just Works" technology, which enables managed code to call API functions in exactly the same way that unmanaged code does, without the marshaling required by other .NET languages. This versatility means that Visual C++ .NET code that calls API functions can often

be simpler and more concise than equivalent code in Visual Basic .NET or Visual C# .NET.

Documentation

The Windows API functions are in various DLLs and libraries whose documentation is spread among several areas in the Windows DDK and Platform SDK. Functions related to detecting devices are in *setupapi.dll* and are documented in the Windows DDK under Device Installation and also in the Platform SDK under Base Services > Device Management. Functions relating to opening communications with devices are in *kernel32.dll* and are documented in the Platform SDK under Base Services in Storage > File Management and in Device Input and Output Control. Functions relating to device notifications are in *user32.dll* and are documented in the Platform SDK under Base Services > Device Management.

The header files for the DLLs often have useful comments as well. A function's documentation typically names the header file. If not, a quick way to find it is to use Windows' *Search* > *For Files or Folders* utility available from the Start menu. In the text box for file names, enter **.h*, and in the text box for words or phrases, enter the name of the function whose declaration you want to find. Be sure that *Include Subfolders* is checked, and let Windows go to work finding the file for you.

Using Visual C++ .NET

To use an API function, a Visual C++ application needs three things: the ability to locate the file containing the function's compiled code, a function declaration, and a call that causes the function to execute.

Each DLL has two or more companion files, a library file (*setupapi.lib, kernel32.lib, user32.lib*) and one or more header files (*setupapi.h, kernel32.h, user32.h*). The library file eliminates the need for the application to get a pointer to the function in the DLL. The header file contains the prototypes, structures, and symbols for the functions that applications may call.

A DLL contains compiled code for the functions the DLL exports, or makes available to applications. For each exported function, the DLL's library file

contains a stub function whose name and arguments match the name and arguments of one of the DLL's functions. The stub function calls its corresponding function in the DLL. During the compile process, the linker incorporates the code in the library file into the application's executable file. When the application calls a function in the library file, the function of the same name in the DLL executes.

The DLLs included with Windows are typically stored in the *%SystemRoot%\system32* folder. Windows searches this folder when an application calls a DLL function. The library and header files for Windows API functions are included in the Windows DDK.

To include a API function in an application, you need to do the following:

1. Add the library files to the project. In Visual Studio, click Project > Properties > Linker > Input. In the *Additional Dependencies* box enter the names of the *.lib* files. If needed, you can enter a path for the library files in the Linker > General window under *Additional Library Directories*.

2. Include the header files in one of the application's files. Here is an example:

```
extern "C" {
#include "hidsdi.h"
#include <setupapi.h>
}
```

The `#include` directive causes the contents of the named file to be included in the file, the same as if they were copied and pasted into the file. The `extern "C"` modifier enables a C++ module to include header files that use C naming conventions. The difference is that C++ uses name decoration, also called name mangling, on external symbols.

To add a path to an include directory, in Visual Studio, click Project > Properties > Resources > General. In the *Additional Include Directories* box enter the path(s) to your *.h* files. (On a command line, these paths are in the compiler's */I* option.)

The punctuation around the file name determines where the compiler will search for the file, and in what order. This is relevant if you have different versions of a file in multiple locations! Enclosing the file name in brackets

(`<setupapi.h>`) causes the compiler to search for the file first in the path specified by the compiler's */I* option, then in the paths specified by the Include environment variable. Enclosing the file name in quotes (`"hidsdi.h"`) causes the compiler to search for the file first in the same directory as the file containing the `#include` directive, then in the directories of any files that contain `#include` directives for that file, then in the path specified by the compiler's */I* option, and finally in the paths specified by the Include environment variable.

The header files for many functions are included automatically when you create a project. For example, *afxwin.h* adds headers for common Windows and MFC functions.

3. Call the function. Here is code that declares the variable `HidGuid` and passes a pointer to it in the function `HidD_GetHidGuid` in *hid.dll*:

```
GUID    HidGuid;
HidD_GetHidGuid(&HidGuid);
```

Using Visual Basic .NET

To use an API function in a Visual Basic .NET program, you need three things: the DLL containing the function, a declaration that enables the application to find the function, and a call that causes the function to execute.

Compared to Visual C++, Visual Basic .NET has additional considerations when calling API functions. The information in the C include files must be translated to Visual-Basic syntax and data types, and the managed .NET code often requires marshaling to enable accessing the unmanaged data returned by an API function.

Instead of a C include file, a Visual Basic .NET application must have Visual-Basic declarations for a DLL's functions and structures. Visual Basic requires references only to the DLLs, not to the library files.

The code to call an API function (or any function in a DLL) follows the same syntax rules as the code to call other Visual-Basic functions. But instead of placing the function's executable code in a routine within the

application, the application requires only a declaration that enables Windows to find the DLL containing the function's code.

Microsoft's documentation for API functions uses C syntax to show how to declare and call the functions. To use an API function in Visual Basic, you need to translate the declaration and function call from C to Visual Basic. The process is more complicated than simple syntax changes, mainly because many of the variable and structure types don't have exact equivalents in Visual Basic.

The Declaration

This is a Visual-Basic declaration for the API function RegisterDeviceNotification, which applications can use to request to be informed when a device is attached or removed:

```
<DllImport("user32.dll", CharSet:=CharSet.Auto)> _
Function RegisterDeviceNotification _
  (ByVal hRecipient As IntPtr, _
  ByVal NotificationFilter As IntPtr, _
  ByVal Flags As Int32) _
  As IntPtr
End Function
```

The declaration contains this information:

- A DllImport attribute that names the file that contains the function's executable code (*user32.dll*). The optional CharSet field is set to CharSet.Auto to cause the operating system to select ANSI (8-bit) or Unicode (16-bit) characters according to the target platform. ANSI is the default for Windows 98 and Windows Me. Unicode is the default for Windows 2000 and Windows XP.
- The function's name (RegisterDeviceNotification).
- The parameters the function will pass to the operating system (hRecipient, NotificationFilter, Flags).
- The data types of the values passed (IntPtr, Int32).
- Whether the parameters will be passed by value (ByVal) or by reference (ByRef). All three parameters in this declaration are passed ByVal.

- The data type of the value returned for the function (IntPtr). A few API calls have no return value and may be declared as subroutines rather than functions.

The declaration must be in the Declarations section of a module.

Providing the DLL's Name

Each declaration must name the file that contains the function's executable code. The file is a DLL. When the application runs, Windows loads the named DLLs into memory (unless they're already loaded).

In most cases, the declaration only has to provide the file name and not the location. The DLLs containing Windows API functions are stored in standard locations (such as *%SytemRoot%\system32*) that Windows searches automatically. For some system files, such as *kernel32.dll*, the *.dll* extension is optional in the declaration.

Data Types

A Visual Basic .NET application can use Visual Basic's data types or their equivalent data types in the .NET Framework. For example, Visual Basic's Integer type is equivalent to a System.Int32 in the .NET Framework.

The C header files for API calls often use additional data types defined in the Platform SDK but not explicitly defined by Visual Basic. So creating a Visual-Basic declaration often requires additional translating. To specify a variable type for an API call, in many cases all you need to do is determine the variable's length, then use a Visual-Basic type that matches. For example, a DWORD is a 32-bit integer, so a Visual-Basic .NET application can declare a DWORD as an Integer. An LPDWORD is a pointer to a DWORD, and can be declared as an Integer passed by reference. A parameter defined in C as a HANDLE can use the System.IntPtr type, which is an Integer with a platform-specific size. A GUID translates to the System.Guid type.

ByRef and ByVal

In calling a function, you can pass the arguments, or parameters, by reference (ByRef) or by value (ByVal). Often either will work. But the concept is important to understand when calling API functions, because many of the functions have variables that must be passed a specific way.

ByRef and ByVal determine what information the call passes to enable the function to access the variable. Every variable has an address in memory where the variable's value is stored. When passing a variable to a function, an application can pass the variable's address or the value itself. The information is passed by placing it on the stack (a temporary storage location).

Passing a variable ByRef means that the function call places the address of the variable on the stack. If the function changes the value by writing a new value to the address, the new value will be available to the calling application because the value will be stored at the address where the application expects to find the variable. The address passed is called a pointer, because it points to, or indicates, the address where the value is stored.

Passing a variable ByVal means that the function call places the value of the variable on the stack. The value at the variable's original address in memory is unchanged. If the function changes the value, the calling application won't know about the change because the function has no way to pass the new value back to the application.

Passing ByVal is the default under Visual Basic .NET. If you want to pass a parameter ByRef, you must specify it in the declaration. (Passing ByRef is the default in Visual Basic 6.)

Except for strings, you must pass a variable ByRef if the called function changes the value and the calling application needs to use the new value. Passing ByRef enables the calling application to access the new value.

Strings are a special case and should be passed ByVal to API functions. If you pass a string ByVal to an API function, Visual Basic actually passes a pointer to the string, as if the string had been declared ByRef. If the function will change the contents of the string, the application should initialize the string to be at least as long as the longest expected returned string.

Passing Structures

Some API functions pass and return structures, which contain multiple items that may be of different types. The documentation for the API functions also documents the structures that the functions pass. The header files contain declarations for the structures in C syntax.

A Visual Basic .NET application can usually declare an equivalent structure in a structure or a class. To ensure that the managed and unmanaged code agree on the layout and alignment of the structure's members, a structure's declaration or class definition can set the StructLayout attribute to LayoutKind.Sequential:

```
<StructLayout(LayoutKind.Sequential)>
```

As with function declarations, the CharSet attribute can determine whether strings are converted to ANSI or Unicode before passing the strings to unmanaged code:

```
<(CharSet:=CharSet.Auto)>
```

A structure can be passed to an API function ByVal, or the application can pass a pointer to the structure using ByRef.

Some structures are difficult or impractical to duplicate in Visual Basic. A solution is to use a generic buffer of the expected size. The application can fill the buffer before passing it and extract returned data from the buffer as needed.

Calling a Function

After declaring a function and any structures or classes to be passed, an application can call the function. This is a call to the RegisterDeviceNotification function declared earlier:

```
Public Const DEVICE_NOTIFY_WINDOW_HANDLE As Integer _
  = 0

deviceNotificationHandle = _
  RegisterDeviceNotification _
  (formHandle, _
  DevBroadcastDeviceInterfaceBuffer, _
  DEVICE_NOTIFY_WINDOW_HANDLE)
```

The DEVICE_NOTIFY_WINDOW_HANDLE constant is defined in *dbt.h*. The formHandle and DevBroadcastDeviceInterfaceBuffer parameters are IntPtr variables. The function returns an IntPtr in deviceNotification-Handle.

Finding Your Device

The Windows API provides a series of SetupDi_ API functions that enable applications to find all devices in a device interface class and to obtain a device path name for each device. The CreateFile function can use the device path name to obtain a handle for accessing the device.

Obtaining a device path name requires these steps:

1. Obtain the device interface GUID.

2. Request a pointer to a device information set with information about all installed and present devices in the device interface class.

3. Request a pointer to a structure that contains information about a device interface in the device information set.

4. Request a structure containing a device interface's device path name.

5. Extract the device path name from the structure.

The application can then use the device path name to open a handle for communicating with the device.

Table 10-1 lists the API functions that applications can use to perform these actions. The functions can be useful for finding devices that use some vendor-specific drivers and HID-class devices that perform vendor-specific functions. For many devices that perform standard functions, applications have other ways to find and gain access to devices. For example, to access a drive, the .NET Framework's Directory class includes a GetLogicalDrives method that enables applications to find all of the logical drives on a system (whether or not they use USB). You can then use methods of the Directory and File classes to access files on the drives.

Table 10-1: Applications use these functions to find devices and obtain device path names to enable accessing devices.

API Function	DLL	Purpose
HidD_GetHidGuid	hid	Retrieve the device interface GUID for the HID class
SetupDiDestroyDeviceInfoList	setupapi	Free resources used by SetupDiGetClassDevs.
SetupDiGetClassDevs	setupapi	Retrieve a device information set for the devices in a specified class.
SetupDiGetDeviceInterfaceDetail	setupapi	Retrieve a device path name.
SetupDiEnumDeviceInterfaces	setupapi	Retrieve information about a device in a device information set.

The following code shows how to use API functions to find a device and obtain its device path name. For complete Visual C++ .NET and Visual Basic .NET applications that demonstrate how to use these functions, go to *www.Lvr.com*.

Obtaining the Device Interface GUID

As Chapter 8 explained, for many drivers, applications can obtain a device interface GUID from a C header file or Visual-Basic declaration provided with the driver. For the HID class, Windows provides an API function to obtain the GUID, which is also defined in *hidclass.h*.

Visual C++

This is the function's declaration:

```
VOID
  HidD_GetHidGuid(
    OUT LPGUID HidGuid
    );
```

This is the code to call the function:

```
HidD_GetHidGuid(&HidGuid);
```

Visual Basic

The function has no return value, so it's declared as a Sub:

```
<DllImport("hid.dll")>
Sub HidD_GetHidGuid _
    (ByRef HidGuid As System.Guid)
End Sub
```

This is the code to call the function:

```
Dim HidGuid As System.Guid
HidD_GetHidGuid(HidGuid)
```

Requesting a Pointer to a Device Information Set

The SetupDiGetClassDevs function can return a pointer to an array of structures containing information about all devices in the device interface class specified by a GUID.

Visual C++

This is the function's declaration:

```
HDEVINFO
  SetupDiGetClassDevs(
    IN LPGUID  ClassGuid,  OPTIONAL
    IN PCTSTR  Enumerator,  OPTIONAL
    IN HWND  hwndParent,  OPTIONAL
    IN DWORD  Flags
    );
```

This is the code to call the function:

```
HANDLE DeviceInfoSet;

DeviceInfoSet = SetupDiGetClassDevs
  (&HidGuid,
  NULL,
  NULL,
  DIGCF_PRESENT|DIGCF_INTERFACEDEVICE);
```

Visual Basic

This is the function's declaration:

```
<DllImport("setupapi.dll", CharSet:=CharSet.Auto)> _
Function SetupDiGetClassDevs _
    (ByRef ClassGuid As System.Guid, _
    ByVal Enumerator As String, _
    ByVal hwndParent As Integer, _
    ByVal Flags As Integer) _
    As IntPtr
End Function
```

This is the code to call the function:

```
Public Const DIGCF_PRESENT As Short = &H2S
Public Const DIGCF_DEVICEINTERFACE As Short = &H10S

Dim DeviceInfoSet As IntPtr

DeviceInfoSet = SetupDiGetClassDevs _
    (HidGuid, _
    vbNullString, _
    0, _
    DIGCF_PRESENT Or DIGCF_DEVICEINTERFACE)
```

Details

For HID-class devices, the ClassGuid parameter is the HidGuid value returned by HidD_GetHidGuid. For other drivers, the application can pass a reference to the appropriate GUID. The Enumerator and hwndParent parameters are unused in this example. The Flags parameter consists of system constants defined in *setupapi.h*. The flags in this example tell the function to look only for device interfaces that are currently present (attached and enumerated) and that are members of the device interface class identified by the ClassGuid parameter.

The value returned, DeviceInfoSet, is a pointer to a device information set that contains information about all attached and enumerated devices in the specified device interface class. The device information set contains a device information element for each device in the set. Each device information ele-

ment contains a handle to a device's devnode (a structure that represents the device) and a linked list of device interfaces associated with the device.

When finished using the device information set, the application should free the resources used by calling SetupDiDestroyDeviceInfoList, as described later in this chapter.

Identifying a Device Interface

A call to SetupDiEnumDeviceInterfaces retrieves a pointer to a structure that identifies a specific device interface in the previously retrieved DeviceInfoSet array. The call specifies a device interface by passing an array index. To retrieve information about all of the device interfaces, an application can loop through the array, incrementing the array index until the function returns zero, indicating that there are no more interfaces. The GetLastError API function then returns *No more data is available*.

How do you know if a device interface is the one you're looking for? The application may need to request more information before deciding to use a device interface. On detecting multiple interfaces, the application can investigate each in turn until finding the desired device or determining that the device isn't present.

Visual C++

This is the declaration for DeviceInterfaceData's type:

```
typedef struct _SP_DEVICE_INTERFACE_DATA {
    DWORD cbSize;
    GUID InterfaceClassGuid;
    DWORD Flags;
    ULONG_PTR Reserved;
} SP_DEVICE_INTERFACE_DATA,
*PSP_DEVICE_INTERFACE_DATA;
```

This is the function's declaration:

```
BOOLEAN
  SetupDiEnumDeviceInterfaces(
    IN HDEVINFO  DeviceInfoSet,
    IN PSP_DEVINFO_DATA  DeviceInfoData,  OPTIONAL
    IN LPGUID  InterfaceClassGuid,
    IN DWORD  MemberIndex,
   OUT PSP_DEVICE_INTERFACE_DATA  DeviceInterfaceData
    );
```

And this is the code to call the function:

```
BOOLEAN Result;
SP_DEVICE_INTERFACE_DATA MyDeviceInterfaceData;

MyDeviceInterfaceData.cbSize =
  sizeof(MyDeviceInterfaceData);
MemberIndex = 0;

Result=SetupDiEnumDeviceInterfaces
  (DeviceInfoSet,
  0,
  &HidGuid,
  MemberIndex,
  &MyDeviceInterfaceData);
```

Visual Basic

This is the declaration for the DeviceInterfaceData structure:

```
<StructLayout(LayoutKind.Sequential)> _
Public Structure SP_DEVICE_INTERFACE_DATA
    Dim cbSize As Integer
    Dim InterfaceClassGuid As System.Guid
    Dim Flags As Integer
    Dim Reserved As Integer
End Structure
```

This is the function's declaration:

```
<DllImport("setupapi.dll")> _
Function SetupDiEnumDeviceInterfaces _
    (ByVal DeviceInfoSet As IntPtr, _
    ByVal DeviceInfoData As Integer, _
    ByRef InterfaceClassGuid As System.Guid, _
    ByVal MemberIndex As Integer, _
    ByRef DeviceInterfaceData As _
        SP_DEVICE_INTERFACE_DATA) _
    As Boolean
End Function
```

This is the code to call the function:

```
Dim MemberIndex As Integer
Dim MyDeviceInterfaceData As SP_DEVICE_INTERFACE_DATA
Dim Result As Boolean

myDeviceInterfaceData.cbSize = _
    Marshal.SizeOf(myDeviceInterfaceData)
MemberIndex = 0

Result = SetupDiEnumDeviceInterfaces _
    (DeviceInfoSet, _
    0, _
    HidGuid, _
    MemberIndex, _
    MyDeviceInterfaceData)
```

Details

In the SP_DEVICE_INTERFACE_DATA structure, the parameter cbSize is the size of the structure in bytes. Before calling SetupDiEnumDeviceInterfaces, the size must be stored in the structure that the function will pass. The sizeof operator in Visual C++ or the Marshal.SizeOf method in Visual Basic retrieves the size. The other values in the structure should be zero.

The HidGuid and DeviceInfoSet parameters are values retrieved previously. DeviceInfoData is an optional pointer to an SP_DEVINFO_DATA structure that limits the search to a particular device instance. MemberIndex is an index to a structure in the DeviceInfoSet array. MyDeviceInterfaceData is

the returned SP_DEVICE_INTERFACE_DATA structure that identifies a device interface of the requested type.

Requesting a Structure Containing the Device Path Name

The SetupDiGetDeviceInterfaceDetail function returns a structure that contains a device path name for a device interface identified in an SP_DEVICE_INTERFACE_DATA structure.

Before calling this function for the first time, there's no way to know the value of the DeviceInterfaceDetailDataSize parameter, which must contain the size in bytes of the DeviceInterfaceDetailData structure. Yet the function won't return the structure unless the function call contains this information. The solution is to call the function twice. The first time, GetLastError returns the error *The data area passed to a system call is too small,* but the RequiredSize parameter contains the correct value for DeviceInterfaceDetailDataSize. The second time, you pass the returned size value and the function returns the structure.

Visual C++

This is the declaration for DeviceInterfaceDetailData's structure:

```
typedef struct _SP_DEVICE_INTERFACE_DETAIL_DATA {
    DWORD cbSize;
    TCHAR DevicePath[ANYSIZE_ARRAY];
} SP_DEVICE_INTERFACE_DETAIL_DATA,
*PSP_DEVICE_INTERFACE_DETAIL_DATA;
```

This is the function's declaration:

```
BOOLEAN
  SetupDiGetDeviceInterfaceDetail(
    IN HDEVINFO  DeviceInfoSet,
    IN PSP_DEVICE_INTERFACE_DATA  DeviceInterfaceData,
    OUT PSP_DEVICE_INTERFACE_DETAIL_DATA
      DeviceInterfaceDetailData,  OPTIONAL
    IN DWORD  DeviceInterfaceDetailDataSize,
    OUT PDWORD  RequiredSize,  OPTIONAL
    OUT PSP_DEVINFO_DATA  DeviceInfoData  OPTIONAL
    );
```

This is the code to call the function the first time:

```
BOOLEAN Result;
PSP_DEVICE_INTERFACE_DETAIL_DATA DetailData;
ULONG Length;

Result = SetupDiGetDeviceInterfaceDetail
  (DeviceInfoSet,
  &MyDeviceInterfaceData,
  NULL,
  0,
  &Length,
  NULL);
```

The code then allocates memory for the DetailData structure, sets the cbSize property of DetailData, and calls the function again, passing the returned buffer size in Length:

```
DetailData =
  (PSP_DEVICE_INTERFACE_DETAIL_DATA)malloc(Length);

DetailData -> cbSize =
  sizeof(SP_DEVICE_INTERFACE_DETAIL_DATA);

Result = SetupDiGetDeviceInterfaceDetail
  (DeviceInfoSet,
  &MyDeviceInterfaceData,
  DetailData,
  Length,
  &Length,
  NULL);
```

Visual Basic

The Visual-Basic code doesn't explicitly declare an SP_DEVICE_INTERFACE_DETAIL_DATA structure for the DeviceInterfaceDetailData parameter. Instead, the code reserves a generic buffer, passes a pointer to the buffer, and extracts the device path name directly from the buffer. So the application doesn't use the following declaration, but I've included it to show what the returned buffer will contain:

```
<StructLayout(LayoutKind.Sequential)> _
Public Structure SP_DEVICE_INTERFACE_DETAIL_DATA
    Dim cbSize As Integer
    Dim DevicePath As String
End Structure
```

This is the function's declaration:

```
<DllImport("setupapi.dll", CharSet:=CharSet.Auto)> _
Function SetupDiGetDeviceInterfaceDetail _
    (ByVal DeviceInfoSet As IntPtr, _
    ByRef DeviceInterfaceData _
        As  SP_DEVICE_INTERFACE_DATA, _
    ByVal DeviceInterfaceDetailData As IntPtr, _
    ByVal DeviceInterfaceDetailDataSize As Integer, _
    ByRef RequiredSize As Integer, _
    ByVal DeviceInfoData As IntPtr) _
    As Boolean
End Function
```

This is the code for the first call:

```
Dim BufferSize As Integer
Dim Success As Boolean

Success = SetupDiGetDeviceInterfaceDetail _
    (DeviceInfoSet, _
    MyDeviceInterfaceData, _
    IntPtr.Zero, _
    0, _
    BufferSize, _
    IntPtr.Zero)
```

After calling SetupDiGetDeviceInterfaceDetail, BufferSize contains the value to pass in the DeviceInterfaceDetailDataSizebuffer parameter in the

next call. But before calling the function again, we need to take care of a few things.

The function will return a pointer (DetailDataBuffer) to an SP_DEVICE_INTERFACE_DETAIL_DATA structure in unmanaged memory. The Marshal.AllocGlocal method uses the returned BufferSize value to reserve memory for the structure:

```
Dim DetailDataBuffer As IntPtr
DetailDataBuffer = Marshal.AllocHGlobal(BufferSize)
```

The cbSize member of the structure passed in DetailDataBuffer equals four bytes for the cbSize integer plus the length of one character for the device path name (which is empty when passed to the function). The Marshal.WriteInt32 method copies the cbSize value into the first member of DetailDataBuffer:

```
Marshal.WriteInt32 _
    (DetailDataBuffer, _
    4 + Marshal.SystemDefaultCharSize)
```

The second call to SetupDiGetDeviceInterfaceDetail passes the DetailDataBuffer pointer and sets the DeviceInterfaceDetailDataSize parameter equal to the BufferSize value returned previously in RequiredSize:

```
Success = SetupDiGetDeviceInterfaceDetail _
    (deviceInfoSet, _
    MyDeviceInterfaceData, _
    DetailDataBuffer, _
    BufferSize, _
    BufferSize, _
    IntPtr.Zero)
```

When the function returns, DetailDataBuffer points to a structure containing a device path name.

Extracting the Device Path Name

The device path name is in the DevicePath member of the SP_DEVICE_INTERFACE_DETAIL_DATA structure returned by SetupDiGetDeviceInterfaceDetail.

Visual C++

The device path name is in DetailData -> DevicePath.

Visual Basic

The string containing the device path name is stored beginning at byte 5 in DetailDataBuffer. (The first four bytes are the cbSize member.) The pDevicePathName variable points to this location:

```
Dim DevicePathName(127) As String

Dim pDevicePathName As IntPtr = _
    New IntPtr(DetailDataBuffer.ToInt32 + 4)
```

The Marshal.PtrToString method retrieves the string from the buffer:

```
DevicePathName = _
    Marshal.PtrToStringAuto(pDevicePathName)
```

We're finished with DetailDataBuffer, so we should free the memory previously allocated for it:

```
Marshal.FreeHGlobal(DetailDataBuffer)
```

Closing Communications

When finished using the DeviceInfoSet returned by SetupDiGetClassDevs, the application should call SetupDiDestroyDeviceInfoList.

Visual C++

This is the function's declaration:

```
BOOL SetupDiDestroyDeviceInfoList(
  HDEVINFO DeviceInfoSet);
```

This is the code to call the function:

```
SetupDiDestroyDeviceInfoList(DeviceInfoSet);
```

Visual Basic

This is the function's declaration:

```
<DllImport("setupapi.dll")> Function
SetupDiDestroyDeviceInfoList _
        (ByVal DeviceInfoSet As IntPtr) _
        As Integer
    End Function
```

This is the code to call the function:

```
SetupDiDestroyDeviceInfoList (deviceInfoSet)
```

Obtaining a Handle

An application can use a retrieved device path name to obtain a handle that enables communicating with the device. Table 10-2 shows the API functions related to requesting a handle.

Requesting a Communications Handle

After retrieving a device path name, an application is ready to open communications with the device. The CreateFile function requests a handle to an object, which can be a file or another resource managed by a driver that supports handle-based operations. For example, applications can request a handle to use in exchanging reports with HID-class devices.

Visual C++

This is the function's declaration:

```
HANDLE CreateFile(
  LPCTSTR lpFileName,
  DWORD dwDesiredAccess,
  DWORD dwShareMode,
  LPSECURITY_ATTRIBUTES lpSecurityAttributes,
  DWORD dwCreationDisposition,
  DWORD dwFlagsAndAttributes,
  HANDLE hTemplateFile
);
```

Table 10-2: Applications can use CreateFile to request a handle to a device and CloseHandle to free the resources used by a handle.

API Function	DLL	Purpose
CloseHandle	kernel32	Free resources used by CreateFile.
CreateFile	kernel32	Retrieve a handle for communicating with a device.

This is the code to call the function:

```
HANDLE DeviceHandle;

DeviceHandle=CreateFile
  (DetailData->DevicePath,
  GENERIC_READ|GENERIC_WRITE,
  FILE_SHARE_READ|FILE_SHARE_WRITE,
  (LPSECURITY_ATTRIBUTES)NULL,
  OPEN_EXISTING,
  0,
  NULL);
```

Visual Basic

This is a declaration for the the SECURITY_ATTRIBUTES structure:

```
<StructLayout(LayoutKind.Sequential)> _
Public Structure SECURITY_ATTRIBUTES
    Dim nLength As Integer
    Dim lpSecurityDescriptor As Integer
    Dim bInheritHandle As Integer
End Structure
```

This is the function's declaration:

```
<DllImport("kernel32.dll", CharSet:=CharSet.Auto)>
Function CreateFile _
    (ByVal lpFileName As String, _
    ByVal dwDesiredAccess As Integer, _
    ByVal dwShareMode As Integer, _
    ByRef lpSecurityAttributes As _
        SECURITY_ATTRIBUTES, _
    ByVal dwCreationDisposition As Integer, _
    ByVal dwFlagsAndAttributes As Integer, _
    ByVal hTemplateFile As Integer) _
    As Integer
End Function
```

This is the code to call the function:

```
Public Const GENERIC_READ = &H80000000
Public Const GENERIC_WRITE = &H40000000
Public Const FILE_SHARE_READ = &H1
Public Const FILE_SHARE_WRITE = &H2
Public Const OPEN_EXISTING = 3
Dim DeviceHandle As Integer
Dim Security As SECURITY_ATTRIBUTES

Security.lpSecurityDescriptor = 0
Security.bInheritHandle = CInt(True)
Security.nLength = Len(Security)

DeviceHandle = CreateFile _
    (DevicePathName, _
    GENERIC_READ Or GENERIC_WRITE, _
    FILE_SHARE_READ Or FILE_SHARE_WRITE, _
    Security, _
    OPEN_EXISTING, _
    0, _
    0)
```

Details

The function passes a pointer to the device-path-name string returned by SetupDiGetDeviceInterfaceDetail. The dwDesiredAccess parameter requests read/write access to the device. The dwShareMode parameter allows other processes to access the device while the handle is open. The lpSecu-

rityAttributes parameter is a pointer to a SECURITY_ATTRIBUTES structure. The dwCreationDisposition parameter must be OPEN_EXISTING for devices. The final two parameters are unused in this example.

Closing the Handle

Chapter 13 shows how to use a handle to exchange information with a HID-class device. For other device classes, the details will vary with the driver. When finished communicating with a device, the application should call CloseHandle to free the resources used by CreateFile.

Visual C++

This is the function's declaration:

```
BOOL CloseHandle(
  HANDLE hObject);
```

This is the code to call the function:

```
CloseHandle(DeviceHandle);
```

Visual Basic

This is the function's declaration:

```
<DllImport("kernel32.dll")> Function CloseHandle _
    (ByVal hObject As Integer) _
    As Integer
End Function
```

This is the code to call the function:

```
CloseHandle(DeviceHandle)
```

Detecting Attachment and Removal

Many applications find it useful to know when a device has been attached or removed. An application that detects when a device has been attached can begin communicating automatically on attachment. An application that detects when a device has been removed can stop attempting to communi-

cate, notify the user, and wait for reattachment. Windows provides device-notification functions for this purpose.

About Device Notifications

To request to be informed when a device is attached or removed, an application's form can register to receive notification messages for devices in a device interface class. The messages are WM_DEVICECHANGE messages that the operating system passes to the form's WindowProc (WndProc in Visual Basic) method. An application can override the WindowProc method in a form's base class with a method that processes the messages and then passes them to the base class's WindowProc method. Each notification contains a device path name that the application can use to identify the device the notification applies to. Table 10-3 lists the API functions used in registering for device notifications. The example that follows shows how to use the functions.

Registering for Device Notifications

Applications use the RegisterDeviceNotification function to request to receive notification messages. The function requires a pointer to a handle for the window or service that will receive the notifications, a pointer to a DEV_BROADCAST_DEVICEINTERFACE structure that holds information about the request, and flags to indicate whether the handle is for a window or service status.

In the DEV_BROADCAST_DEVICEINTERFACE structure passed to RegisterDeviceNotification, the dbcc_devicetype member is set to DBT_DEVTYP_DEVICEINTERFACE to specify that the application wants to receive notifications about a device interface class, and classguid is the GUID of the device interface class (HidGuid in the examples).

When the WM_DEVICECHANGE messages are no longer of interest, the application should call UnregisterDeviceNotification, as described later in this chapter.

Table 10-3: These functions enable an application to request to receive or stop receiving notifications about device attachment and removal.

API Function	DLL	Purpose
RegisterDeviceNotification	user32	Request to receive device notifications
UnregisterDeviceNotification	user32	Request to stop receiving device notifications

Visual C++

The declaration for the DEV_BROADCAST_DEVICEINTERFACE structure is this:

```
typedef struct _DEV_BROADCAST_DEVICEINTERFACE {
    DWORD dbcc_size;
    DWORD dbcc_devicetype;
    DWORD dbcc_reserved;
    GUID dbcc_classguid;
    TCHAR dbcc_name[1];
} DEV_BROADCAST_DEVICEINTERFACE
*PDEV_BROADCAST_DEVICEINTERFACE;
```

This is the function's declaration:

```
HDEVNOTIFY RegisterDeviceNotification(
  HANDLE hRecipient,
  LPVOID NotificationFilter,
  DWORD Flags
);
```

This is the code to call the function:

```
HDEVNOTIFY DeviceNotificationHandle;

DEV_BROADCAST_DEVICEINTERFACE
  DevBroadcastDeviceInterface;

DevBroadcastDeviceInterface.dbcc_size =
  sizeof(DevBroadcastDeviceInterface);

DevBroadcastDeviceInterface.dbcc_devicetype =
  DBT_DEVTYP_DEVICEINTERFACE;

DevBroadcastDeviceInterface.dbcc_classguid = HidGuid;
```

```
DeviceNotificationHandle = RegisterDeviceNotification
  (m_hWnd,
  &DevBroadcastDeviceInterface,
  DEVICE_NOTIFY_WINDOW_HANDLE);
```

Visual Basic

The device-notification functions use several constants defined in header files. These are from *dbt.h*:

```
Public Const DBT_DEVTYP_DEVICEINTERFACE As Integer = 5
Public Const DEVICE_NOTIFY_WINDOW_HANDLE As Integer _
    = 0
Public Const WM_DEVICECHANGE As Integer = &H219
```

These are from *setupapi.h*:

```
Public Const DIGCF_PRESENT As Short = &H2S
Public Const DIGCF_DEVICEINTERFACE As Short = &H10S
```

The DEV_BROADCAST_DEVICEINTERFACE structure has this declaration:

```
 <StructLayout(LayoutKind.Sequential)> _
 Public Class DEV_BROADCAST_DEVICEINTERFACE
     Public dbcc_size As Integer
     Public dbcc_devicetype As Integer
     Public dbcc_reserved As Integer
     Public dbcc_classguid As Guid
     Public dbcc_name As Short
End Class
```

This is the declaration for RegisterDeviceNotification:

```
<DllImport("user32.dll", CharSet:=CharSet.Auto)> _
Function RegisterDeviceNotification _
    (ByVal hRecipient As IntPtr, _
    ByVal NotificationFilter As IntPtr, _
    ByVal Flags As Int32) _
    As IntPtr
End Function
```

This is the code to call the function:

```
Dim DevBroadcastDeviceInterface _
    As DEV_BROADCAST_DEVICEINTERFACE = _
        New DEV_BROADCAST_DEVICEINTERFACE()
Dim DevBroadcastDeviceInterfaceBuffer As IntPtr
Dim DeviceNotificationHandle As IntPtr
Dim Size As Integer
Friend frmMy As frmMain
```

The Marshal.SizeOf method retrieves the size of the DEV_BROADCAST_DEVICEINTERFACE structure, which is then stored in the structure's dbcc_size member:

```
Size = Marshal.SizeOf(DevBroadcastDeviceInterface)

DevBroadcastDeviceInterface.dbcc_size = Size
DevBroadcastDeviceInterface.dbcc_devicetype = _
    DBT_DEVTYP_DEVICEINTERFACE
DevBroadcastDeviceInterface.dbcc_reserved = 0
DevBroadcastDeviceInterface.dbcc_classguid = _
    HidGuid
```

Marshal.AllocGlobal reserves memory for a buffer that will hold the DEV_BROADCAST_DEVICEINTERFACE structure. The Marshal.StructureToPointer method copies the structure into the buffer. The application is then ready to call RegisterDeviceNotification:

```
DevBroadcastDeviceInterfaceBuffer = _
    Marshal.AllocHGlobal(Size)

Marshal.StructureToPtr _
    (DevBroadcastDeviceInterface, _
    DevBroadcastDeviceInterfaceBuffer, _
    True)

DeviceNotificationHandle = _
    RegisterDeviceNotification _
    (frmMy.Handle, _
    DevBroadcastDeviceInterfaceBuffer, _
    DEVICE_NOTIFY_WINDOW_HANDLE)
```

When finished using DevBroadcastDeviceInterfaceBuffer, the application should free the memory allocated for it by AllocHGlobal:

```
Marshal.FreeHGlobal _
    (DevBroadcastDeviceInterfaceBuffer)
```

Capturing Device Change Messages

The WindowProc function processes messages received by a form, dialog box, or other window.

Visual C++

To receive WM_DEVICECHANGE messages, a dialog box's message map must contain the line ON_WM_DEVICECHANGE():

```
BEGIN_MESSAGE_MAP(MyApplicationDlg, CDialog)
  //{{AFX_MSG_MAP(MyApplicationDlg)
  .
  .
  .
  //}}AFX_MSG_MAP
  ON_WM_DEVICECHANGE()
END_MESSAGE_MAP()
```

Visual Basic

This is the code for a WndProc routine that overrides the base form's default WndProc routine:

```
Protected Overrides Sub WndProc(ByRef m As Message)

    If m.Msg = WM_DEVICECHANGE Then
        OnDeviceChange(m)
    End If

    MyBase.WndProc(m)

End Sub
```

On receiving a WM_DEVICECHANGE message, the method calls the OnDeviceChange method and then passes the message to the WndProc method in the form's base class.

Reading Device Change Messages

On receiving a WM_DEVICECHANGE message, a window's OnDeviceChange method executes. The method can examine the message's contents and take any needed action. The message contains two pointers: lParam and wParam.

The wParam property is a code that indicates device arrival, removal, or another event.

The lParam property is a device management structure. There are several types of device-management structures, but all begin with the same header, which has three members. The header is a DEV_BROADCAST_HDR structure whose dbch_devicetype member indicates the type of device-management structure that lParam points to.

If dbch_devicetype = DBT_DEVTYP_DEVICEINTERFACE, the structure is a DEV_BROADCAST_INTERFACE and the application can retrieve the complete structure, read the device path name in the dbcc_name member, and compare the name to the device path name of the device of interest.

Visual C++

This is the declaration for the DEV_BROADCAST_HDR structure:

```
typedef struct _DEV_BROADCAST_HDR {
    DWORD dbch_size;
    DWORD dbch_devicetype;
    DWORD dbch_reserved;
} DEV_BROADCAST_HDR, *PDEV_BROADCAST_HDR;
```

This is the code for the OnDeviceChange function:

```
BOOL CUsbhidiocDlg::OnDeviceChange
  (WPARAM wParam,
  LPARAM lParam)
{
  switch(wParam)
  {
    case DBT_DEVICEARRIVAL:
      // Find out if the device path name matches
      // wParam.
      // If yes, perform any tasks required
      // on device attachment.

        return TRUE;

    case DBT_DEVICEREMOVECOMPLETE:

      // Find out if the device path name matches
      // wParam.
      // If yes, perform any tasks required
      // on device removal.

      return TRUE;

    default:
      return TRUE;
  }
}
```

Visual Basic

These constants are from dbt.h:

```
Public Const DBT_DEVICEARRIVAL As Integer = &H8000
Public Const DBT_DEVICEREMOVECOMPLETE As Integer _
    = &H8004
```

This is the declaration for the DEV_BROADCAST_HDR structure:

```
<StructLayout(LayoutKind.Sequential)> _
Public Class DEV_BROADCAST_HDR
    Public dbch_size As Integer
    Public dbch_devicetype As Integer
    Public dbch_reserved As Integer
End Class
```

This is code to check for device arrival and removal messages:

```
Friend Sub OnDeviceChange(ByVal m as Message)

    If (m.WParam.ToInt32 = DBT_DEVICEARRIVAL) Then

        ' Find out if the device path name matches
        ' wParam.
        ' If yes, perform any tasks required
        ' on device removal.

    ElseIf (m.WParam.ToInt32 = _
        DBT_DEVICEREMOVECOMPLETE) Then

        ' Find out if the device path name matches
        ' wParam.
        ' If yes, perform any tasks required
        ' on device removal.

    End If

End Sub
```

Retrieving the Device Path Name in the Message

If the message indicates a device arrival or removal (or another event of interest), the application can investigate further.

In the structure that lParam points to, if dbch_devicetype contains DBT_DEVTYP_DEVICEINTERFACE, the event relates to a device interface. The structure in lParam is a DEV_BROADCAST_INTERFACE structure, which begins with a DEV_BROADCAST_HDR structure. The dbcc_name member contains the device path name of the device the message applies to.

The application can compare this device path name with the device path name of the device of interest. On a match, the application can take any desired actions.

Visual C++

This is the code to retrieve the device path name and look for a match:

```
PDEV_BROADCAST_HDR lpdb = (PDEV_BROADCAST_HDR)lParam;

if (lpdb->dbch_devicetype ==
  DBT_DEVTYP_DEVICEINTERFACE)
{
  PDEV_BROADCAST_DEVICEINTERFACE lpdbi =
    (PDEV_BROADCAST_DEVICEINTERFACE)lParam;

  CString DeviceNameString;

  DeviceNameString = lpdbi->dbcc_name;

  if
   ((DeviceNameString.CompareNoCase
       (DetailData>DevicePath)) == 0)
  {
    // The names match.
  }
  else
  {
    // It's a different device.
  }
}
```

Visual Basic

The application uses two declarations for the DEV_BROADCAST_DEVICEINTERFACE structure. The first declaration, presented earlier, is used when calling RegisterDeviceNotification. The second declaration, DEV_BROADCAST_DEVICEINTERFACE_1, enables marshaling the data in dbcc_name and classguid:

```
<StructLayout _
    (LayoutKind.Sequential, _
    CharSet:=CharSet.Unicode)> _
Public Class DEV_BROADCAST_DEVICEINTERFACE_1
    Public dbcc_size As Integer
    Public dbcc_devicetype As Integer
    Public dbcc_reserved As Integer
    <MarshalAs _
        (UnmanagedType.ByValArray, _
        ArraySubType:=UnmanagedType.U1, _
        SizeConst:=16)> _
        Public dbcc_classguid() As Byte
    <MarshalAs _
        (UnmanagedType.ByValArray, sizeconst:=255)> _
        Public dbcc_name() As Char
    End Class
```

This is the code to retrieve the device path name and look for a match:

```
Dim DevBroadcastDeviceInterface As _
    New DEV_BROADCAST_DEVICEINTERFACE_1()
Dim DevBroadcastHeader As New DEV_BROADCAST_HDR()

Marshal.PtrToStructure(m.LParam, DevBroadcastHeader)

If (DevBroadcastHeader.dbch_devicetype = _
    DBT_DEVTYP_DEVICEINTERFACE) Then
    Dim StringSize As Integer = _
        CInt((DevBroadcastHeader.dbch_size - 32) / 2)
    ReDim DevBroadcastDeviceInterface.dbcc_name _
            (StringSize)

    Marshal.PtrToStructure _
        (m.LParam, DevBroadcastDeviceInterface)
```

```
        Dim DeviceNameString As New String _
            (DevBroadcastDeviceInterface.dbcc_name, _
            0, _
            StringSize)

        If (String.Compare _
            (DeviceNameString, _
            DevicePathName, _
            True) = 0) Then
                'The name matches.
        Else
                'It's a different device.
        End If
    End If
```

MarshalPtrToStructure copies the message's lParam property into a DEV_BROADCAST_HDR structure. If lParam indicates that the message relates to a device interface, the application retrieves the device path name.

The name is in a Char array in unmanaged memory. The application needs to retrieve the Char array and convert it to a String.

The dbch_size member of DEV_BROADCAST_HDR contains the number of bytes in the complete DEV_BROADCAST_INTERFACE structure. To obtain the number of characters in the device path name stored in dbch_name, subtract the 32 bytes in the structure that are not part of the name and divide by 2 because there are 2 bytes per character.

DevBroadcastDeviceInterface is a DEV_BROADCAST_INTERFACE_1 structure that marshals the data in the classguid and dbcc_name members. A ReDim statement trims dbcc_name to match the size of the device path name. Marshal.PtrToStructure copies the data from the unmanaged block in lParam to the DevBroadcastDeviceInterface structure. The Char array containing the device path name is then stored as a String in DeviceNameString, and the String.Compare method looks for a match.

Stopping Device Notifications

To stop receiving device notifications, an application calls UnregisterDeviceNotification. The application should call the function before closing.

Visual C++

This is the function's declaration:

```
BOOL UnregisterDeviceNotification(
  HDEVNOTIFY Handle
);
```

This is the code to call the function:

```
UnregisterDeviceNotification(
  DeviceNotificationHandle);
```

Visual Basic

This is the function's declaration:

```
<DllImport("user32.dll")> Function
UnregisterDeviceNotification _
    (ByVal Handle As IntPtr) _
    As Boolean
    End Function
```

This is the code to call the function:

```
UnregisterDeviceNotification _
    (DeviceNotificationHandle)
```