13

Human Interface Devices: Host Application

Chapter 10 showed how to obtain a handle to communicate with a device. This chapter shows how Visual Basic .NET and Visual C++ .NET applications can use the handle to communicate with a HID-class device.

HID API Functions

The Windows API includes functions that applications can use to learn about a HID's reports and to send and receive report data. The Windows DDK documents the functions.

The HID API considers each report item to be either a button or value. As defined by the HID API, a button is a control or data item that has a discrete, binary value, such as ON (1) or OFF (0). Buttons include items repre-

sented by unique Usage IDs in the Buttons, Keyboard, and LED Usage pages. A value usage, or value, can have any of a range of values. Any report item that isn't a button is a value.

Windows 98 Gold was the first to include the HID API. Windows 98 SE, Windows 2000, and Windows Me support additional HID functions. The API was expanded again for Windows XP. The tables in this chapter note the functions that aren't available in all Windows editions.

Requesting Information about the HID

Table 13-1 lists API functions that request information about a HID and its reports. HidD_GetPreparsedData retrieves a pointer to a buffer that contains information about the HID's reports. HidP_GetCaps uses the pointer to retrieve a HIDP_CAPS structure that specifies what report types a device supports and provides information about the type of information in the reports. For example, the structure includes the number of HIDP_BUTTON_CAPS structures that have information about a button or set of buttons. The application can then use the HidP_GetButtonCaps function to retrieve these structures

The API also includes several functions for retrieving strings. Table 13-2 lists these.

Sending and Receiving Reports

Table 13-3 lists functions that applications can use to send and receive reports.

Windows' HID driver causes the host controller to request Input reports periodically. The driver stores received reports in a buffer. ReadFile retrieves one or more reports from the buffer. If the buffer is empty, ReadFile waits for a report to arrive. ReadFile thus does not cause the device to send a report but just reads reports that the driver has requested.

WriteFile sends an Output report to the HID. WriteFile uses an interrupt transfer if the HID has an interrupt OUT endpoint and the operating system is later than Windows 98 Gold. Otherwise, WriteFile uses a control

Table 13-1: Applications can use these API functions to obtain information about a HID and its reports.

API Function	Purpose
HidD_FreePreparsedData	Free resources used by HidD_GetPreparsedData.
HidD_GetPhysicalDescriptor[1]	Retrieve a physical descriptor.
HidD_GetPreparsedData	Return a handle to a buffer with information about the HID's reports.
HidP_GetButtonCaps	Retrieve an array with information about the buttons in a top-level collection for a specified report type.
HidP_GetCaps	Retrieve a structure describing a HID's reports.
HidP_GetExtendedAttributes[1]	Retrieve a structure with information about Global items the HID parser didn't recognize.
HidP_GetLinkCollectionNodes	Retrieve a structure with information about collections within a top-level collection.
HidP_GetSpecificButtonCaps	Like HidP_GetButtonCaps but can specify a Usage Page, Usage ID, and link collection.
HidP_GetSpecificValueCaps	Like HidP_GetValueCaps but can specify a Usage Page, Usage ID, and link collection.
HidP_GetValueCaps	Retrieve an array with information about the values in a top-level collection for a specified report type.
HidP_IsSameUsageAndPage	Determine if two Usages (Usage Page and Usage ID) are equal.
HidP_MaxDataListLength	Retrieve the maximum number of HIDP_DATA structures that HidP_GetData can return for a HID report type and top-level collection.
HidP_MaxUsageListLength	Retrieve the maximum number of Usage IDs that HidP_GetUsages can return for a report type and top-level collection.
HidP_ TranslateUsagesToI8042 ScanCodes	Map Usages on the HID_USAGE_PAGE_KEYBOARD Usage Page to PS/2 scan codes.
HidP_ UsageAndPageListDifference	Retrieve the differences between two arrays of Usages (Usage Page and Usage ID).
HidP_UsageListDifference	Retrieve the differences between two arrays of Usage IDs.

[1]Not supported under Windows 98 Gold.

Table 13-2: Applications can use these API functions to retrieve strings from a HID.

API Function	Purpose
HidD_GetIndexedString[1]	Retrieve a specified string.
HidD_GetManufacturerString[1]	Retrieve a manufacturer string
HidD_GetProductString[1]	Retrieve a product string.
HidD_GetSerialNumberString[1]	Retrieve a serial-number string.
[1]Not supported under Windows 98 Gold.	

transfer with a Set_Report request. If using interrupt transfers, WriteFile will wait if the device NAKs. If using control transfers, WriteFile returns with an error code on failure or a timeout.

HidD_GetInputReport requests an Input report using a control transfer with a Get_Report request, bypassing the Input report buffer. HidD_SetOutputReport provides a way to send an Output report using a control transfer with a Set_Report request, even if the HID and operating system support using interrupt transfers.

For Feature reports, HidD_GetFeature retrieves a report using a control transfer and Get_Report request and HidD_SetFeature sends a report using a control transfer and Set_Report request. Note that HidD_SetFeature is not the same as the standard USB request Set_Feature!

All of the functions that use control transfers return with an error code on failure or a timeout.

Providing and Using Report Data

After retrieving a report, an application can use the raw data directly from the buffer or use API functions to extract button or value data. In a similar way, an application can write data to be sent directly into a report's buffer or use API functions to place the data into a buffer for sending.

Table 13-4 lists API functions that extract information in received reports and store information in reports to be sent. For example, an application can

Table 13-3: Applications can use these API functions to send and receive reports.

API Function	Purpose
HidD_GetFeature	Read a Feature report.
HidD_GetInputReport[1]	Read an Input report using a control transfer.
HidD_SetFeature	Send a Feature report.
HidD_SetOutputReport[1]	Send an Output report using a control transfer.
ReadFile	Read an Input report obtained via an interrupt transfer.
WriteFile	Send an Output report. Use an interrupt transfer if possible, otherwise use a control transfer.
[1]Requires Windows XP or later.	

find out what buttons have been pressed by calling HidP_GetButtons, which returns a buffer containing the Usage IDs of all buttons that belong to a specified Usage Page and are set to ON. An application can set and clear buttons in a report to be sent by calling HidP_SetButtons and HidP_UnsetButtons. In a similar way, applications can retrieve and set values in a report using HidP_GetUsageValue and Hid_Set_UsageValue.

Managing HID Communications

Table 13-5 lists API functions that applications can use in managing HID communications.

Chapter 10 showed how to use HidD_GetHidGuid to obtain the device interface GUID for the HID class. HidD_SetNumInputBuffers enables an application to change the size of the HID driver's buffer for Input reports. A larger buffer can be helpful if the application might be too busy at times to read reports before the buffer overflows. The value set is the number of reports the buffer will hold. HidD_FlushQueue deletes any Input reports in the buffer.

Identifying a Device

After obtaining a handle to a HID as described in Chapter 10, an application can use the HID API functions to find out whether the HID is one that

Table 13-4: Applications can use these API functions to extract information in retrieved reports and store information in reports to be sent.

API Function	Purpose
HidP_GetButtons	Same as HidP_GetUsages.
HidP_GetButtonsEx	Same as HidP_GetUsagesEx.
HidP_GetData	Retrieve an array of structures, with each structure identifying either the data index and state of a button control that is set to ON (1) or the data index and data for a value control.
HidP_GetScaledUsageValue	Retrieve a signed and scaled value from a report.
HidP_GetUsages	Retrieve a list of all of the buttons that are on a specified Usage Page and are set to ON (1).
HidP_GetUsagesEx	Retrieve a list of all of the buttons that are set to ON (1).
HidP_GetUsageValue	Retrieve the data for a specified value.
HidP_GetUsageValueArray	Retrieve data for an array of values with the same Usage ID.
HidP_InitializeReportForID[1]	Set all buttons to OFF (0) and set all values to their null values if defined and otherwise to zero.
HidP_SetButtons	Same as HidP_SetUsages.
HidP_SetData	Sets the states of buttons and data in values in a report.
HidP_SetScaledUsageValue	Convert a signed and scaled physical number to a Usage's logical value and set the value in a report.
HidP_SetUsages	Set one or more buttons in a report to ON (1).
HidP_SetUsageValue	Set the data for a specified value.
HidP_SetUsageValueArray	Set the data for an array of values with the same Usage ID.
HidP_UnsetButtons	Same as HidP_UnsetUsages.
HidP_UnsetUsages	Set one or more buttons in a report to OFF (0).
[1]Not supported under Windows 98 Gold.	

the application wants to communicate with. The application can identify a device by its Vendor ID and Product ID, or by searching for a device with a specific Usage, such as a game controller.

Reading the Vendor and Product IDs

For vendor-specific devices that don't have standard Usages, searching for a device with a specific Vendor ID and Product ID is often useful. The API

Table 13-5: Applications can use these API functions in managing HID communications.

API Function	Purpose
HidD_FlushQueue	Delete all Input reports in the buffer.
HidD_GetHidGuid	Retrieve the device interface GUID for HID-class devices.
HidD_GetNumInputBuffers[1]	Retrieve the number of reports the Input report buffer can hold.
HidD_SetNumInputBuffers[1]	Set the number of reports the Input report buffer can hold.
HidRegisterMinidriver	HID mini-drivers call this function during initialization to register with the HID class driver.
[1]Not supported under Windows 98 Gold.	

function HidD_GetAttributes retrieves a pointer to a structure containing the Vendor ID, Product ID, and device release number.

Visual C++

The HIDD_ATTRIBUTES structure contains information about the device:

```
typedef struct _HIDD_ATTRIBUTES {
  ULONG   Size;
  USHORT  VendorID;
  USHORT  ProductID;
  USHORT  VersionNumber;
} HIDD_ATTRIBUTES, *PHIDD_ATTRIBUTES;
```

This is the function's declaration:

```
BOOLEAN
  HidD_GetAttributes(
    IN HANDLE HidDeviceObject,
    OUT PHIDD_ATTRIBUTES Attributes
    );
```

This is the code to retrieve the structure:

```
BOOLEAN Result;
HIDD_ATTRIBUTES Attributes;

// Set the Size member to the number of bytes
// in the structure.
Attributes.Size = sizeof(Attributes);
Result = HidD_GetAttributes
  (DeviceHandle,
  &Attributes);
```

The application can then compare the Vendor ID and Product ID to the desired values:

```
const unsigned int VendorID = 0x0925;
const unsigned int ProductID = 0x1234;

if (Attributes.VendorID == VendorID) {
    if (Attributes.ProductID == ProductID) {
        // The Vendor ID and Product ID match.
    }
    else {
        // The Product ID doesn't match.
        // Close the handle.
    }
}
else {
    // The Vendor ID doesn't match.
    // Close the handle.
}
```

Visual Basic

The HIDD_ATTRIBUTES structure contains information about the device:

```
<StructLayout(LayoutKind.Sequential)> _
Public Structure HIDD_ATTRIBUTES
    Dim Size As Integer
    Dim VendorID As Short
    Dim ProductID As Short
    Dim VersionNumber As Short
End Structure
```

This is the declaration for the function:

```
<DllImport("hid.dll")> _
Function HidD_GetAttributes _
    (ByVal HidDeviceObject As Integer, _
    ByRef Attributes As HIDD_ATTRIBUTES) _
    As Boolean
End Function
```

This is the code to retrieve the structure:

```
Dim DeviceAttributes As HIDD_ATTRIBUTES
Dim MyVendorID as Short
Dim MyProductID as Short
Dim Result as BOOLEAN

' Set the Size property of DeviceAttributes to the
' number of bytes in the structure.

DeviceAttributes.Size =
    Marshal.SizeOf(myHID.DeviceAttributes)

Result = HidD_GetAttributes _
    (DeviceHandle, _
    DeviceAttributes)
```

The application can then compare the Vendor ID and Product ID to the desired values:

```
MyVendorID = &h0925
MyProductID = &h1234

If (DeviceAttributes.VendorID = MyVendorID) And _
    (DeviceAttributes.ProductID = MyProductID) Then

    Debug.WriteLine("My device detected")

Else

    Debug.WriteLine("Not my device")
    ' Close the handle.
```

Details

DeviceHandle is a handle returned by CreateFile. Before calling HidD_GetAttributes, the Size member of the DeviceAttributes must be set to the structure's size. If the function returns True, the DeviceAttributes structure filled without error. The application can then compare the retrieved values with the desired Vendor ID and Product ID and device release number.

If the attributes don't indicate the desired device, the application should use the CloseHandle API function to close the handle to the interface. The application can then move on to test the next HID in the device information set retrieved with SetupDiGetClassDevs as described in Chapter 10.

Getting a Pointer to a Buffer with Device Capabilities

Another way to find out more about a device is to examine its capabilities. You can do this for a device whose Vendor ID and Product ID matched the values you were looking for, or you can examine the capabilities for an unknown device.

The first task is to call HidD_GetPreparsedData to get a pointer to a buffer with information about the device's capabilities.

Visual C++

This is the function's declaration:

```
BOOLEAN
  HidD_GetPreparsedData(
    IN HANDLE HidDeviceObject,
    OUT PHIDP_PREPARSED_DATA *PreparsedData
    );
```

This is the code to call the function:

```
PHIDP_PREPARSED_DATA PreparsedData;

HidD_GetPreparsedData
  (DeviceHandle,
  &PreparsedData);
```

Visual Basic

This is the function's declaration:

```
<DllImport("hid.dll")> _
Function HidD_GetPreparsedData _
    (ByVal HidDeviceObject As Integer, _
    ByRef PreparsedData As IntPtr) _
    As Boolean
End Function
```

This is the code to call the function:

```
Dim PreparsedData As IntPtr

HidD_GetPreparsedData _
    (DeviceHandle, _
    PreparsedData)
```

Details

DeviceHandle is the handle returned by CreateFile. PreparsedData is a pointer to the buffer containing the data. The application doesn't need to access the data in the buffer directly. The code just needs to pass the pointer to another API function.

When finished using the PreparsedData buffer, the application should free system resources by calling HidD_FreePreparsedData as described later in this chapter.

Getting the Device's Capabilities

The HidP_GetCaps function returns a pointer to a structure that contains information about the device's capabilities. The structure contains the HID's Usage Pages, Usages, report lengths, and the number of button-capabilities structures, value-capabilities structures, and data indexes that identify specific controls and data items in Input, Output, and Feature reports. An application can use the capabilities information to identify a particular HID and learn about its reports and report data. Not every item in the structure applies to all devices.

Visual C++

This is the declaration for the HIDP_CAPS structure:

```
typedef struct _HIDP_CAPS
{
    USAGE    Usage;
    USAGE    UsagePage;
    USHORT   InputReportByteLength;
    USHORT   OutputReportByteLength;
    USHORT   FeatureReportByteLength;
    USHORT   Reserved[17];

    USHORT   NumberLinkCollectionNodes;

    USHORT   NumberInputButtonCaps;
    USHORT   NumberInputValueCaps;
    USHORT   NumberInputDataIndices;

    USHORT   NumberOutputButtonCaps;
    USHORT   NumberOutputValueCaps;
    USHORT   NumberOutputDataIndices;

    USHORT   NumberFeatureButtonCaps;
    USHORT   NumberFeatureValueCaps;
    USHORT   NumberFeatureDataIndices;
} HIDP_CAPS, *PHIDP_CAPS;
```

This is the function's declaration:

```
NTSTATUS
  HidP_GetCaps(
    IN PHIDP_PREPARSED_DATA PreparsedData,
    OUT PHIDP_CAPS Capabilities
    );
```

This is the code to call the function:

```
HIDP_CAPS Capabilities;

HidP_GetCaps
  (PreparsedData,
  &Capabilities);
```

Visual Basic

This is the declaration for the HIDP_CAPS structure:

```
<StructLayout(LayoutKind.Sequential)> _
Public Structure HIDP_CAPS
    Dim Usage As Short
    Dim UsagePage As Short
    Dim InputReportByteLength As Short
    Dim OutputReportByteLength As Short
    Dim FeatureReportByteLength As Short
    <MarshalAs _
        (UnmanagedType.ByValArray, _
        SizeConst:=17)> _
        Dim Reserved() As Short
    Dim NumberLinkCollectionNodes As Short
    Dim NumberInputButtonCaps As Short
    Dim NumberInputValueCaps As Short
    Dim NumberInputDataIndices As Short
    Dim NumberOutputButtonCaps As Short
    Dim NumberOutputValueCaps As Short
    Dim NumberOutputDataIndices As Short
    Dim NumberFeatureButtonCaps As Short
    Dim NumberFeatureValueCaps As Short
    Dim NumberFeatureDataIndices As Short
End Structure
```

This is the declaration for the function:

```
<DllImport("hid.dll")> _
Function HidP_GetCaps _
    (ByVal PreparsedData As IntPtr, _
    ByRef Capabilities As HIDP_CAPS) _
    As Boolean
End Function
```

This is the code to call the function:

```
Dim Capabilities As HIDP_CAPS

HidP_GetCaps _
    (PreparsedData, _
    Capabilities)
```

Details

PreparsedData is the pointer returned by HidD_GetPreparsedData. When the function returns, the application can examine and use whatever values are of interest in the Capabilities structure. For example, if you're looking for a joystick, you can look for a Usage Page of 01h and a Usage of 04h.

The report lengths are useful for setting buffer sizes for sending and receiving reports.

Getting the Capabilities of the Buttons and Values

The device capabilities aren't the only thing that an application can retrieve from a HID. The application can also get the capabilities of each button and value in a report.

HidP_GetValueCaps returns a pointer to an array of structures containing information about the values in a report. The NumberInputValueCaps property of the HIDP_CAPS structure is the number of structures returned by HidP_GetValueCaps.

The items in the structures include many values obtained from the HID's report descriptor, as described in Chapter 12. The items include the Report ID, whether a value is absolute or relative, whether a value has a null state, and logical and physical minimums and maximums. A LinkCollection identifier distinguishes between controls with the same Usage and Usage Page in the same collection.

In a similar way, the HidP_GetButtonCaps function can retrieve information about a report's buttons. The information is stored in a HidP_ButtonCaps structure.

An application that has no use for this information doesn't have to retrieve it.

Sending and Receiving Reports

All of the previous API functions are concerned with finding and learning about a device that matches what the application is looking for. On finding

Table 13-6: The transfer type used to send or receive a report can vary with the API function, operating system edition, and available endpoints.

Report Type	API Function	Transfer Type
Input	ReadFile	Interrupt IN
	HidD_GetInputReport	Control with Get_Report request
Output	WriteFile	Interrupt OUT if possible; otherwise Control with Set_Report request
	HidD_SetOutputReport	Control with Set_Report request
Feature IN	HidD_GetFeature	Control with Get_Report request
Feature OUT	HidD_SetFeature	Control with Set_Report request

a device of interest, the application and device are ready to exchange data in reports.

Table 13-3 listed the six API functions for exchanging reports. Table 13-6 shows that the transfer type the host uses varies with the report type and may also vary depending on the operating system and available endpoints.

Sending an Output Report to the Device

On obtaining a handle and learning the number of bytes in the report, an application can send an Output report to the HID. The application copies the data to send to a buffer and calls WriteFile. As Chapter 11 explained, the type of transfer the HID driver uses to send the Output report depends on the Windows edition and whether the HID interface has an interrupt OUT endpoint. The application doesn't have to know or care which transfer type the driver uses.

Visual C++

This is the function's declaration:

```
BOOL WriteFile(
  HANDLE hFile,
  LPCVOID lpBuffer,
  DWORD nNumberOfBytesToWrite,
  LPDWORD lpNumberOfBytesWritten,
  LPOVERLAPPED lpOverlapped
);
```

This the code to call the function:

```
BOOLEAN Result;

// The report data can reside in a byte array.
// The array size should equal at least the report
// length in bytes + 1.

CHAR OutputReport[3];

DWORD BytesWritten;

// The first byte in the buffer containing the report
// is the Report ID.

OutputReport[0]=0;

// Store data to send in OutputReport[] in the
// bytes following the Report ID.
// Example:

OutputReport[1]=79;
OutputReport[2]=75;

Result = WriteFile
  (DeviceHandle,
  OutputReport,
  Capabilities.OutputReportByteLength,
  &BytesWritten,
  NULL);
```

Visual Basic

This is the function's declaration:

```
<DllImport("kernel32.dll")> Function WriteFile _
    (ByVal hFile As Integer, _
    ByRef lpBuffer As Byte, _
    ByVal nNumberOfBytesToWrite As Integer, _
    ByRef lpNumberOfBytesWritten As Integer, _
    ByVal lpOverlapped As Integer) _
    As Boolean
End Function
```

This is the code to send an Output report to the HID:

```
Dim NumberOfBytesWritten As Integer
Dim OutputReportBuffer() As Byte
Dim ReportID as Integer
Dim Result as Boolean

ReDim OutputReportBuffer _
        (Capabilities.OutputReportByteLength - 1)

ReportID = 0
OutputReportBuffer(0) = ReportID

' Store data to send in OutputReportBuffer()
' in the bytes following the report ID.
' Example:

OutputReportBuffer(1) = 79
OutputReportBuffer(2) = 75

Result = WriteFile _
    (DeviceHandle, _
    OutputReportBuffer(0), _
    UBound(OutputReportBuffer) + 1, _
    NumberOfBytesWritten, _
    0)
```

Details

The hFile parameter is the handle returned by CreateFile. The lpBuffer parameter points to the buffer that contains the report data. The nNumberOfBytesToWrite parameter specifies how many bytes to write and

should equal the OutputReportByteLength property of the HIDP_CAPS structure retrieved with HidP_GetCaps. This value equals the report size in bytes plus one byte for the Report ID, which is the first byte in the buffer. The buffer must be large enough to hold the Report ID and report data.

The lpOverlapped parameter is unused in this example, but WriteFile can use overlapped I/O as described in the following section on ReadFile. Overlapped I/O can prevent the application's thread from hanging if the HID's interrupt OUT endpoint NAKs endlessly. In normal operation, the endpoint should accept received data with little delay.

On success, the function returns True with NumberOfBytesWritten containing the number of bytes the function successfully wrote to the HID.

If the interface supports only the default Report ID of 0, the Report ID doesn't transmit on the bus, but the Report ID must always be present in the buffer the application passes to WriteFile.

When sending a report to an interrupt endpoint, WriteFile returns on success or an error. If the device NAKs the report data, WriteFile waits until the endpoint ACKs the data. When sending a report via the control endpoint, WriteFile returns on success, an error, or a timeout (if the endpoint continues to NAK the report data).

Probably the most common error returned by WriteFile in HID communications is *CRC Error*. This error indicates that the host controller attempted to send the report, but the device didn't respond as expected. In spite of the error message, the problem isn't likely to be due to an error detected in a CRC calculation. The error is more likely to be due to a firmware problem that is keeping the endpoint from accepting the report data. If WriteFile doesn't return at all, the interrupt OUT endpoint probably has not been configured to accept the report data.

Reading an Input Report from the Device

The complement to WriteFile is ReadFile. When the application has a handle to the HID interface and knows the number of bytes in the device's

Input report, the application can use ReadFile to read an Input report from a device.

To read a report, the application declares a buffer to hold the data and calls ReadFile. The buffer size should equal at least the size reported in the InputReportByteLength property of the HIDP_CAPS structure returned by HidP_GetCaps.

When called with non-overlapped I/O, ReadFile is a blocking call. If an application calls ReadFile when the HID's read buffer is empty, the application's thread waits until either a report is available, the user closes the application from the Task Manager, or the user removes the device from the bus. There are several approaches to keeping an application from hanging as it waits for a report. The device can continuously send reports. The application can attempt to read a report only after requesting one using an Output or Feature report. The application can use ReadFile with overlapped I/O and a timeout. The ReadFiles can also take place in a separate thread.

To ensure that the device always has data to send, you can write the firmware so that the IN endpoint is always enabled and ready to respond to a request for data. If there is no new data to send, the device can send the same data as last time, or the device can return a vendor-defined code that indicates there is nothing new to report. Or before each ReadFile, the application can send a report that prompts the firmware to provide a report to send to the host.

In an overlapped read, ReadFile returns immediately even if there is no report available, and the application can call WaitForSingleObject to retrieve the report. The advantage of WaitForSingleObject is the ability to set a timeout. If the data hasn't arrived when the timeout period has elapsed, the function returns a code that indicates a timeout and the application can try again or use the CancelIo function to cancel the read operation. This approach works well if reports are normally available without delay, but the application needs to regain control if for some reason there is no report.

To prevent long delays waiting for WaitForSingleObject to return, an application can set the timeout to zero and call the function repeatedly in a loop or periodically, triggered by a timer. The function returns immediately if no

report is available, and the application can perform other tasks in the loop or between timeouts.

Another way to improve the performance of an application that is reading Input reports is to do the ReadFiles in a separate thread that notifies the main thread when a report is available. A .NET application can define an asynchronous delegate and use the BeginInvoke method to call a method that performs the ReadFiles in a different thread. BeginInvoke can specify a callback routine that executes in the application's main thread when the method that has called ReadFile returns, enabling the application to retrieve the returned report.

Visual C++

In addition to CreateFile, introduced in Chapter 10, an overlapped ReadFile uses these functions:

```
BOOL CancelIo
  (HANDLE hFile);

HANDLE CreateEvent
  (LPSECURITY_ATTRIBUTES lpEventAttributes,
  BOOL bManualReset,
  BOOL bInitialState,
  LPCTSTR lpName);

BOOL ReadFile
  (HANDLE hFile,
  LPVOID lpBuffer,
  DWORD nNumberOfBytesToRead,
  LPDWORD lpNumberOfBytesRead,
  LPOVERLAPPED lpOverlapped);

DWORD WaitForSingleObject
  (HANDLE hHandle,
  DWORD dwMilliseconds);
```

This is the code for doing an overlapped ReadFile:

```
CHAR        InputReportBuffer[3];
DWORD       BytesRead;
DWORD       Result;
HANDLE      hEventObject;
OVERLAPPED HIDOverlapped;

hEventObject = CreateEvent
  ((LPSECURITY_ATTRIBUTES)NULL,
  FALSE,
  TRUE,
  "");

HIDOverlapped.hEvent = hEventObject;
HIDOverlapped.Offset = 0;
HIDOverlapped.OffsetHigh = 0;

// Set the first byte in the buffer to the Report ID.
InputReportBuffer[0] = 0;

ReadHandle=CreateFile
  (DetailData->DevicePath,
  GENERIC_READ|GENERIC_WRITE,
  FILE_SHARE_READ|FILE_SHARE_WRITE,
  (LPSECURITY_ATTRIBUTES)NULL,
  OPEN_EXISTING,
  FILE_FLAG_OVERLAPPED,
  NULL);

Result = ReadFile
  (ReadHandle,
  InputReportBuffer,
  Capabilities.InputReportByteLength,
  &BytesRead,
  (LPOVERLAPPED) &HIDOverlapped);

Result = WaitForSingleObject
  (hEventObject,
  3000);
```

```
switch (Result)
{
    case WAIT_OBJECT_0: {

        // Success;
        // Use the report data;

        break;
        }
    case WAIT_TIMEOUT: {

        // Timeout error;
        //Cancel the read operation.

        CancelIo(ReadHandle);
        break;
        }
    default: {

        // Undefined error;
        //Cancel the read operation.

        CancelIo(ReadHandle);
        break;
        }
    }
```

Visual Basic

These are the constants and structures used in an overlapped ReadFile:

```
Public Const FILE_FLAG_OVERLAPPED As Integer _
    = &H40000000
Public Const FILE_SHARE_READ As Short = &H1S
Public Const FILE_SHARE_WRITE As Short = &H2S
Public Const GENERIC_READ As Integer = &H80000000
Public Const GENERIC_WRITE As Integer = &H40000000
Public Const OPEN_EXISTING As Short = 3
Public Const WAIT_OBJECT_0 As Short = 0
Public Const WAIT_TIMEOUT As Integer = &H102
```

```
<StructLayout(LayoutKind.Sequential)> _
Public Structure OVERLAPPED
    Dim Internal As Integer
    Dim InternalHigh As Integer
    Dim Offset As Integer
    Dim OffsetHigh As Integer
    Dim hEvent As Integer
End Structure
```

In addition to CreateFile, introduced in Chapter 10, an overlapped ReadFile uses these functions:

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

<DllImport("kernel32.dll", CharSet:=CharSet.Auto)> _
Function CreateEvent _
    (ByRef SecurityAttributes _
        As SECURITY_ATTRIBUTES, _
    ByVal bManualReset As Integer, _
    ByVal bInitialState As Integer, _
    ByVal lpName As String) _
    As Integer
End Function

<DllImport("kernel32.dll")> _
Function ReadFile _
    (ByVal hFile As Integer, _
    ByRef lpBuffer As Byte, _
    ByVal nNumberOfBytesToRead As Integer, _
    ByRef lpNumberOfBytesRead As Integer, _
    ByRef lpOverlapped As OVERLAPPED) _
    As Integer
End Function

<DllImport("kernel32.dll")> _
Function WaitForSingleObject _
    (ByVal hHandle As Integer, _
    ByVal dwMilliseconds As Integer) _
    As Integer
End Function
```

This the code to do an overlapped ReadFile:

```
Dim EventObject As Integer
Dim HIDOverlapped As OVERLAPPED
Dim InputReportBuffer() As Byte
Dim NumberOfBytesRead As Integer
Dim Result As Integer
Dim Security As SECURITY_ATTRIBUTES
Dim Success As Boolean

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

EventObject = CreateEvent _
    (Security,
    CInt(False),
    CInt(True),
    "")

HIDOverlapped.Offset = 0
HIDOverlapped.OffsetHigh = 0
HIDOverlapped.hEvent = EventObject

' Set the first byte in the report buffer to the
' report ID.

InputReportBuffer(0) = 0;

ReadHandle = CreateFile _
    (DevicePathName, _
    GENERIC_READ Or GENERIC_WRITE, _
    FILE_SHARE_READ Or FILE_SHARE_WRITE, _
    Security, _
    OPEN_EXISTING, _
    FILE_FLAG_OVERLAPPPED, _
    0)

ReDim InputReportBuffer _
    (Capabilities.InputReportByteLength - 1)
```

```
Result = ReadFile _
    (ReadHandle, _
    InputReportBuffer(0), _
    Capabilities.InputReportByteLength, _
    NumberOfBytesRead, _
    HIDOverlapped)

Result = WaitForSingleObject _
    (EventObject, _
    3000)

Select Case Result
    Case WAIT_OBJECT_0

        ' Success
        ' Use the report data.

    Case WAIT_TIMEOUT

        ' Timeout error.
        ' Cancel the Read operation.

        CancelIo(ReadHandle)

    Case Else

        ' Undefined error.
        ' Cancel the Read operation.

        CancelIo(ReadHandle)

    End Select
```

Details

Before calling ReadFile for the first time, the application calls CreateEvent to create an event object that is set to the signaled state when the ReadFile operation completes. Overlapped I/O requires a handle obtained from a call to CreateFile with the dwFlagsAndAttributes parameter set to FILE_FLAG_OVERLAPPPED.

InputReportBuffer is a byte array that must be large enough to hold the report ID and the largest Input report defined in the HID's report descriptor.

The call to ReadFile passes the handle returned by CreateFile, the address of the first element in the InputReportBuffer array, the report's length from the Capabilities structure returned by HidP_GetCaps, an Integer to hold the number of bytes read, and an overlapped structure whose hEvent parameter is a handle to the event object. A call to ReadFile returns immediately. The application then calls WaitForSingleObject, which returns when a report has been read or on a timeout. The parameters passed to WaitForSingleObject are the event object and a timeout value in milliseconds.

If WaitForSingleObject returns success (WAIT_OBJECT_0), the first byte in InputReportBuffer is the report ID, and the following bytes are the report data read from the device. If the interface supports only the default report ID of zero, the report ID doesn't transmit on the bus but is always present in the buffer returned by ReadFile.

A call to ReadFile doesn't initiate traffic on the bus. The call just retrieves a report that the host previously requested in one of its periodic interrupt IN transfers. If there are no unread reports, ReadFile waits for a report to arrive. The host begins requesting reports when the HID driver is loaded during enumeration. The driver stores the reports in a ring buffer. When the buffer is full and a new report arrives, the oldest report is overwritten. A call to ReadFile reads the oldest report in the buffer. Under Windows 98 SE and later, an application can set the buffer size with the HidD_SetNumInputBuffers function. Different Windows editions have different default buffer sizes, ranging from 2 under Windows 98 Gold to 32 under Windows XP.

Each handle with READ access to the HID has its own Input buffer, so multiple applications can read the same reports.

If the application doesn't request reports as frequently as they're sent, some will be lost. One way to keep from losing reports is to increase the size of the report buffer passed to ReadFile. If multiple reports are available, ReadFile returns as many as will fit in the buffer. If you need to be absolutely sure not

to lose a report, use Feature reports instead. Also see the tips in Chapter 3 about performing time-critical transfers.

The Idle rate introduced in Chapter 11 determines whether or not a device sends a report if the data hasn't changed since the last transfer. During enumeration, Windows' HID driver attempts to set the Idle rate to 0, which means that the HID won't send a report unless the report data has changed. There is no API call that enables applications to change the Idle rate. To prevent setting an Idle rate of zero, the HID can return a STALL to the Set_Idle request to inform the host the request isn't supported. Not all device controllers have hardware support for the Idle rate, though support can be implemented with a timer in firmware.

Whether or not Set_Idle is supported, the firmware can be programmed to send each report only once. After sending a report, the firmware can configure the endpoint to return NAK in response to IN token packets. When the device has new data to send, the firmware can configure the endpoint to send a report. Otherwise, the device will continue to send the same data every time the host polls the endpoint, and the application is likely to read the same report multiple times.

If ReadFile isn't returning, these are possible reasons:

- The HID's interrupt IN endpoint is NAKing the IN token packets because the endpoint hasn't been configured to send the report data. Remember that the endpoint's hardware interrupt typically triggers after data has been sent, so the device must prepare to send the initial report before the first interrupt.
- The number of bytes the endpoint is sending doesn't equal the number of bytes in the report (for the default report ID) or the number of bytes in the report + 1 (for other report IDs).
- The endpoint is sending report ID zero with the report, or the endpoint isn't sending a report ID greater than zero with the report.

Writing a Feature Report to the Device

To send a Feature report to a device, use the HidD_SetFeature function. The function sends a Set_Report request and a report in a control transfer.

Visual C++

This is the function's declaration:

```
BOOLEAN
  HidD_SetFeature(
    IN HANDLE HidDeviceObject,
    IN PVOID ReportBuffer,
    IN ULONG ReportBufferLength
    );
```

This is the code to call the function:

```
CHAR OutFeatureReportBuffer[3];
BOOLEAN Result;

// The first byte in the report buffer is the
// report ID:

OutFeatureReportBuffer[0]=0;

// Store data to send in FeatureReport[] in the
// bytes following the Report ID.
// Example:

OutFeatureReportBuffer[1]=79;
OutFeatureReportBuffer[2]=75;

Result = HidD_SetFeature
  (DeviceHandle,
  OutFeatureReportBuffer,
  Capabilities.FeatureReportByteLength);
```

Visual Basic

This is the function's declaration:

```
<DllImport("hid.dll")> _
Function HidD_SetFeature _
    (ByVal HidDeviceObject As Integer, _
    ByRef lpReportBuffer As Byte, _
    ByVal ReportBufferLength As Integer) _
    As Boolean
End Function
```

This is the code to call the function:

```
Dim OutFeatureReportBuffer _
    (Capabilities.FeatureReportByteLength - 1) as Byte
Dim Success As Boolean

'The first byte in the report buffer is the report ID:

OutFeatureReportBuffer(0) = 0

' Example report data following the report ID:

OutFeatureReportBuffer(1) = 55
OutFeatureReportBuffer(2) = 41

Success = HidD_SetFeature _
    (DeviceHandle, _
    OutFeatureReportBuffer(0), _
    Capabilities.FeatureReportByteLength)
```

Details

A byte array holds the report to send. The first byte in the array is the report ID. The length of the Feature report plus one byte for the report ID is in the HIDP_CAPS structure retrieved by HidP_GetCaps. HidD_SetFeature requires a handle to the HID, the address of the first element in the byte array, and length of the byte array.

The function returns True on success. If the device continues to NAK the report data, the function times out and returns.

A call to HidD_SetOutputReport works much the same way to send an Output report using a control transfer. The function passes a handle to the HID, a pointer to a byte array containing an Output report, and the number of bytes in the report plus one byte for the report ID.

Reading a Feature Report from a Device

To read a Feature report from a device, use the HidD_GetFeature API function. The function sends a Get_Feature request in a control transfer. The device returns the report in the Data stage.

Visual C++

This is the function's declaration:

```
BOOLEAN
  HidD_GetFeature(
    IN HANDLE HidDeviceObject,
    OUT PVOID ReportBuffer,
    IN ULONG ReportBufferLength
    );
```

This is the code to call the function:

```
BOOLEAN Result;
CHAR    InFeatureReportBuffer[3];

// The first byte in the report buffer is the report
// ID:

InFeatureReportBuffer[0]=0;

Result = HidD_GetFeature
  (DeviceHandle,
  InFeatureReportBuffer,
  Capabilities.FeatureReportByteLength)
```

Visual Basic

This is the function's declaration:

```
<DllImport("hid.dll")> Function HidD_GetFeature _
    (ByVal HidDeviceObject As Integer, _
    ByRef lpReportBuffer As Byte, _
    ByVal ReportBufferLength As Integer) _
    As Boolean
End Function
```

This is the code to call the function:

```
Dim InFeatureReportBuffer _
    (Capabilities.FeatureReportByteLength - 1) as Byte
Dim Success As Boolean

'The first byte in the report buffer is the report ID:

InFeatureReportBuffer(0) = 0

Success = HidD_GetFeature _
    (DeviceHandle, _
    InFeatureReportBuffer(0), _
    Capabilities.FeatureReportByteLength)
```

Details

A byte array holds the retrieved report. The first byte in the array is the report ID. The length of the Feature report plus one byte for the report ID is in the HIDP_CAPS structure retrieved by HidP_GetCaps. HidD_GetFeature requires a handle to the HID, the address of the first element in the byte array, and length of the byte array.

The function returns True on success. If the device continues to NAK in the Data stage of the transfer, the function times out and returns.

A call to HidD_GetInputReport works in much the same way to request an Input report using a control transfer. The function passes a handle to the HID, a pointer to a byte array to hold the Input report, and the number of bytes in the report plus one byte for the report ID.

Closing Communications

When finished communicating with a device, the application should call CloseHandle to close any handles opened by CreateFile, as described in Chapter 10. When finished using the PreparsedData buffer returned by HidD_GetPreparsedData, the application should call HidD_FreePreparsedData.

Visual C++

This is declaration for HidD_FreePreparsedData:

```
BOOLEAN
  HidD_FreePreparsedData(
    IN PHIDP_PREPARSED_DATA PreparsedData
    );
```

This is the code to call the function:

```
HidD_FreePreparsedData(PreparsedData);
```

Visual Basic

This is the declaration for HidD_FreePreparsedData:

```
<DllImport("hid.dll")> _
Function HidD_FreePreparsedData _
    (ByRef PreparsedData As IntPtr) _
    As Boolean
End Function
```

This is the code to call the function:

```
HidD_FreePreparsedData(PreparsedData)
```