Working with USB devices in .NET and C#

07 Jul 2010 | by Ashley Deakin | Filed in .NET C#

Comments PDF

Buzz Demo Source Code.zip

This article was originally published in VSJ, which is now part of Developer Fusion.

This article was originally published in VSJ, which is now part of Developer Fusion.

USB devices are now ubiquitous across the hardware space, but there are still no native support within the .NET library in order to interact with them. In this article, we look at creating the .NET software to run on a PC connected to a USB device. We will also use the techniques presented here to control a device that was probably never intended to be used with a PC, but opens up lots of possibilities for fun PC game projects as well as illustrating the portability and simplicity of USB HID devices.

USB overview

A USB device provides information about itself in the form of *descriptors*. A client can then query the device to work out its class and capabilities. Two important properties on all USB devices are its *Vendor ID* (VID) and *Product ID* (PID), which can be used to uniquely identify the device type by the operating system's plug and play system. Each device can support multiple interfaces or *endpoints*. Each device must support endpoint 0 which is reserved for system use such as device enumeration.

The operating system provides logical connections or *pipes* from the client to endpoints on the host device. There are *stream pipes* and *message pipes*. Endpoints are classified into four types depending upon the transfer type supported: control transfers, bulk transfers, interrupt transfers and isochronous transfers. The bulk and isochronous transfers tend to be used for devices that transfer large amounts of data in bursts or streams such as digital cameras or printers. The others are commonly used for more immediate or command/response type communications.

There are many classifications of USB devices, each with a protocol specification of how to use it. These specification documents are available to view online. But amongst all the specific definitions for mass storage devices, monitors and printers, there is one that is much more general.

Introducing HID

The device type in question is the *Human Interface Device* or HID interface (see **Figure 1**).

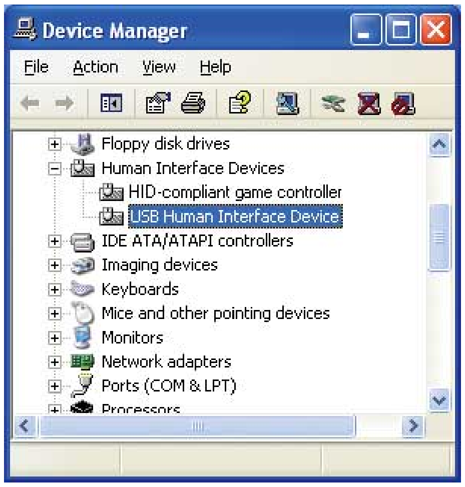

Figure 1: HID device

It is promoted as easy to implement from both the point of view of the driver and of the device. Although it has a specific name, its uses extend far beyond the limits that name may imply. It is commonly associated with keyboards, mice and game controllers, but it is simple to extend for custom devices. Devices with a HID interface only communicate with either interrupt or control pipes. Data is exchanged in the form of *reports*. There are three types of report: input, output and feature.

HID reports are made up of *usages* organised into *usage pages*. A usage describes what a particular portion of a report represents, for example 'X axis', 'Y axis' or 'throttle'. A client ascertains the format of a report during device enumeration. To formally define the reports for a custom device, you can download the HID Descriptor tool.

When a HID device is plugged into a Windows XP PC, the operating system will detect it and try its best to locate a driver for it. But even if no driver is found, Windows device manager will confirm that the device is plugged in and that it is HID compliant. If you select the 'Details' tab, you can see a string that contains details of the device including its VID and PID. You'll need these later when locating it programmatically.

Finding the device

The first step in connecting to the device is to locate it. Of course, as is the nature of a USB device, it can be inserted or removed at any time. Any serious software solution needs to take this into consideration. To detect when a device is inserted and removed, a program must register its interest in such events by sending its main window handle to RegisterDeviceNotification. When a new device is detected or a device removed, Windows sends a WM_DEVICECHANGE message. Override WndProc on a Form to handle the messages. Once notification of a new device has been received, the program needs to search the list of devices currently on the USB bus to check if the device that was inserted is the one it is interested in.

To get the device list, call SetupDiGetClassDevs, one of the USB API calls in setupapi.dll. Before we go into details, I need to tell you about the Windows USB API documentation. Unless you have already installed the Windows DDK, there practically isn't any! This isn't made easier by the fact that there are many USB functions in the API, only a few of which you will ever use. You can find information about the API functions from the Internet, but it is sparse at best.

SetupDiGetClassDevs provides information about all devices of a certain class (e.g. HID) currently connected to the client PC. The first parameter to the function is a GUID that specifies the class of device to look for. Use HidD_GetHidGuid in hid.dll to obtain the GUID that Windows uses to represent the HID class. The function reserves a block of memory (an InfoSet) that holds an array with one entry per device. The entries in the InfoSet have to be enumerated with calls to SetupDiEnumDeviceInterfaces in 'setupapi.dll'. Each call to this function fills a DeviceInterfaceData structure with details about a device in the list with a non-zero return code.

But we're still not there! We have to drill down even further. Now we have to fill a DeviceInterfaceData-Detail structure using a call to SetupDiGetDeviceInterfaceDetail. This finally provides us with a string that represents a path to the device (just like a file path). This path contains the VID and PID of the device amongst other things. You can check portions of this string against the VID and PID of your device. If they match, you've found your device!

```
Device VID 0x054C
Device PID 0x1000
Path \\?\hid#vid_054c
&pid_1000#6&2dd74f76&0&0000#{4d1e55b2-
f16f-11cf-88cb-001111000030}
```

In the above example, the VID and PID can be seen just after the HID specifier (vid_054c&pid_1000). A simple substring match should be sufficient (e.g. string.IndexOf()).

Once all this is complete, you must remember to free the infoset with a call to SetupDiDestroyDevice-InfoList.

Device operations

Armed with the path to the device, it can be opened like a standard file using the CreateFile API call and opening it overlapped. For those unfamiliar with Windows IO, this means that the resulting file supports concurrent asynchronous read and write operations. Once the device is open, reading and writing to it are just like reading and writing to a file, although the number of bytes that can be read or written is limited to the lengths of the output and input reports for the device. You can obtain information about the reports with a call to HidD_GetPreparsedData in hid.dll followed by a call to HidP_GetCaps. The first API call reserves a block of memory that the second accesses. Remember to free the memory with a call to HidD_FreePreparsedData. The structure obtained from GetCaps contains details about the device's capabilities, including the lengths of the input and output reports.

Once the handle to a device has been obtained, it can be passed into the constructor of a FileStream object. Normal read and write operations can be performed on the FileStream, but the real power comes in when using them asynchronously:

```
private void BeginAsyncRead()
{
    byte[] byIn = new
        byte[InputReportLength];
    File.BeginRead(byIn, 0,
        InputReportLength, new
        AsyncCallback(ReadCompleted),
        byIn);
}

protected void ReadCompleted(
    IAsyncResult iResult)
{
    byte[] byIn =
        (byte[])iResult.AsyncState;
    try
    {
        File.EndRead(iResult);
        try
        {
                HandleDataReceived(byIn);
        }
        finally
        {
                BeginAsyncRead();
        }
    }
```

```
        catch (IOException ioexc)
        {
            // Device has been removed!
        }
    }
```

Calling the API functions does require that the appropriate structures and API calls are defined correctly. You should be able to find all that you need on www.pinvoke.net, but the most useful are listed in Table 1.

Table 1 : USB API Calls

Call	Dll	Usage
SetupDiGetClassDevs	setupapi	Gets all connected devices by class
SetupDiDestroyDeviceInfoList	setupapi	Frees memory allocated in above
SetupDiEnumDeviceInterfaces	setupapi	Gets details of a connected device
SetupDiGetDeviceInterfaceDetail	setupapi	Gets further details of above
HidD_GetPreparsedData	hid	Gets details of open HID device
HidD_FreePreparsedData	hid	Frees memory allocated in above
HidP_GetCaps	hid	Gets capabilities of open HID device
HidD_GetHidGuid	hid	Gets GUID for HID device class

Now the fun part

Having described the mechanics, now we'll put it into practice. In this section, you'll get an idea of just how easy it is to program for simple devices. I will show you how to create a C# 'driver' for the custom controller that is sold with the Sony Playstation 2 game 'Buzz' (published by Sony Computer Entertainment Europe). The game costs around £29.99 at Amazon, and is terrific at parties. For those of you who haven't seen it, Buzz is an entertaining and original game based around a TV music quiz show. You can play solo or with up to four players, each having a hand held buzzer (see **Figure 2**) that is used to input the responses to the game questions, usually against the clock.

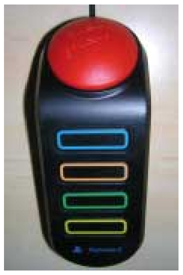

Figure 2: The Buzz controller

There are four handsets connected like a hydra through a central moulded hub with a single USB connector. The are five coloured buttons on each handset and the big round, red one has a lamp which can be turned on or off.

When the USB connector is plugged into a port on a PC, it will be detected as a compatible HID device, although Windows will be unable to find any actual driver. Device Manager will confirm that the device is HID compliant and will show its VID and PID, which can be used to locate the device. After detecting, locating and opening the device, we must read and write reports to control the handsets. Unfortunately, information on how the reports are structured is not freely available, but after a bit of experimentation, the appropriate formats become clear and seem remarkably straightforward.

There is one output report used to control the lights on the handsets and one input report giving details of which buttons are currently depressed. To change which handsets are illuminated, simply write an output report with the appropriate bits set. When a user presses or releases one of the buttons on the handset, the device issues an input report. There is no need to poll the device (repeatedly check its state) as the device sends unsolicited input reports whenever a button is pressed or released.

The output reports are formatted as a byte array. The first two bytes are not used and should be ignored. Bytes 3 to 6 control the state of the lamp on each handset (byte 3, handset 1; byte 4, handset 2; etc). Writing a zero turns the lamp off, 0xFF turns it on and anything else leaves it unchanged.

The input reports are slightly more complex. The state of each button is represented by a bit; 1 for pressed and 0 for not pressed. These bits are packed into five bit chunks per handset, with the first bit of each chunk appended immediately after the last bit of the previous chunk. So some bitwise masking is required to extract the button states for each handset. The first button bit is sent in byte 4, so the first three bytes can be ignored. See **Figure 3**.

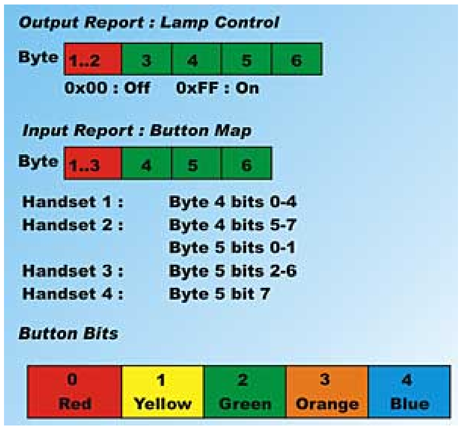

Figure 3: The controller format

Writing the output reports is trivial, so this is a good place to start. Reading the input reports is not quite as easy, and using the .Net Framework's asynchronous reads is recommended. Simply kick off an asynchronous read using a buffer and byte read count the same length as the input report. When a report is read, the read terminates. If an asynchronous read terminates with an exception, the device has been removed.

I'm sure at this point you can envisage many different entertaining applications for which the Buzz handsets could be used. I'll leave developing them up to you.

What USB gives you

Using a USB interface on your device has many benefits including:

- Hot swappable devices
- Powered interface
- High speed data
- Bus architecture
- Standard protocol

Resources

- www.usb.org
- MSDN Coding4Fun – Is That You? Writing Better Software for Cool USB Hardware by Scott Hanselmann,
- USB In a Nutshell
- The HID Page
- C# USB HID Driver
- SharpUSBLib

You might also like...

Contribute

Why not write for us? Or you could submit an event or a user group in your area. Alternatively just tell us what you think!

Web Development

ASP.NET Quickstart
Programming news
Java programming
ASP.NET tutorials
C# programming

Developer Jobs

ASP.NET Jobs
Java Jobs
Developer Jobs

Our tools

We've got automatic conversion tools to convert C# to VB.NET, VB.NET to C#. Also you can compress javascript and compress css and generate sql connection strings.