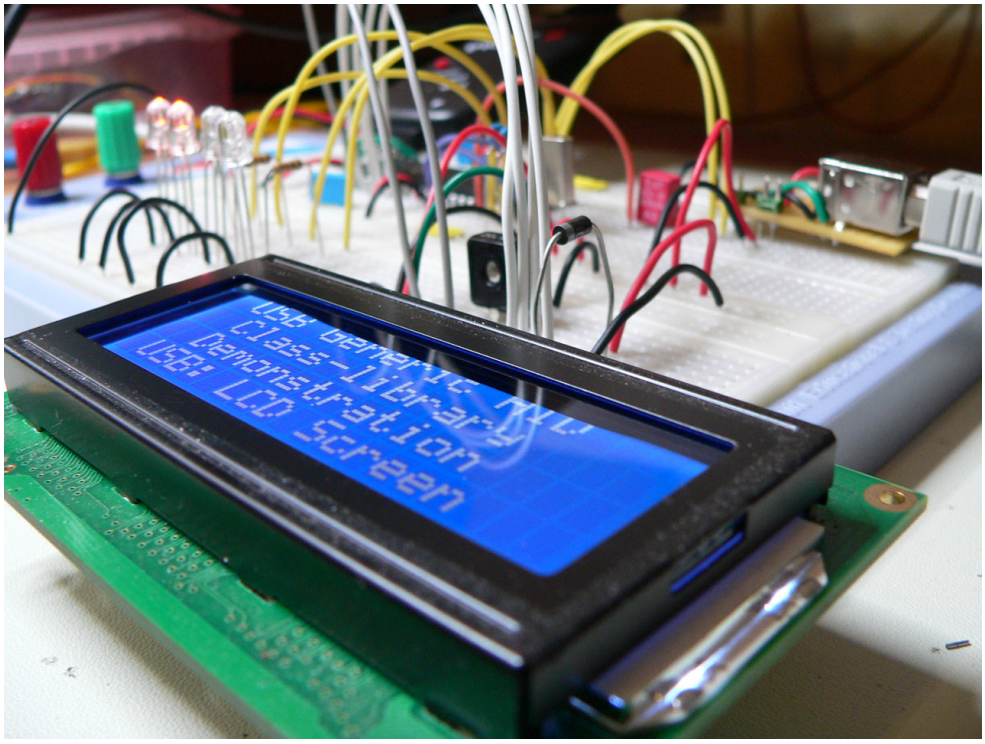
Open Source Framework for USB Generic HID devices based on the PIC18F and Windows

Posted by Simon Inns Posted on April 8, 2010 PIC Microcontroller, USB 7 Comments.

Contents [hide]

If you've dabbled with PIC18F microcontrollers and the USB Generic HID standard before (perhaps you've even tried my Building a PIC18F USB device project) then you will have noticed that there is a lot of complexity in supporting USB on both the PIC18F and the Windows host-side of things.



USB Generic HID device

Getting beyond the basic steps of reading a switch and flashing an LED (how many projects have you built to flash LEDs?!) is actually quite a large task with a steep learning-curve.

To make things easier on the hobbyist who wishes to delve into more exciting projects, I've developed a framework for producing USB devices which covers both the Windows host-side application development and the PIC18F firmware itself. The framework consists of a Visual Studio C# class library (which handles all of the intricacy of dealing with the Windows specific SDKs and operating system requirements), a windows reference application (which demonstrates how to use the class library and acts as a testing front-end for the library), a simple USB hardware reference design and (last but not least!) a PIC18F4550 firmware which communicates with the class library.

In essence this means that you can rapidly develop and test a USB device and the Windows host-application with a minimum of USB Generic HID protocol knowledge. The class library gives you a very simple interface to the USB device from C# and the firmware serves as an example of how to create the specific software needed on the PIC for your device design.

The framework includes the ability to pass commands and responses to and from the USB device as well as bi-directional bulk-transfer of data allowing you to build more complex data-capture applications which require more information to be passed quickly from host to device and vice-versa.

In this article I will go over each element of the framework and show how you can use it to develop your own creative USB devices.

For the impatient, here is a quick YouTube video of the framework in action. The LCD driver example shown in the video can be downloaded below:

Opensource Generic HID USB Framework - PIC18F4550

Version 3_0_0_0

Version 3 of the USB library and firmware is now available (2011-10-16). I've made several major and minor enhancements to the library as well as some bug fixing. Here is a list of the enhancements and changes:

- Firmware
  - Firmware is now targeted for the Microchip C18 compiler (and is no longer compatible with Hi-Tech PICC18)
  - Microchip's Application Library (containing the USB stack) is now separated from the framework code and is installed standalone
  - The firmware now supports version 2.9a of the USB stack
  - Support for USB bootloaders is now included (lightly tested at the moment)
  - Both interrupt driven USB polling and non-interrupt driven USB polling are now supported
  - Debug stream now supports dynamic text so you can place live variables and other information into the in-built debug functions
  - Open-source firmware code now separated correctly from Microchip stack code
  - General code clean-up with better (and more concise) comments and notes
- Host Reference Application
  - Fixed memory leakage issue in non-Managed code (readRawReportFromDevice method)
  - Changed the method used for registering to receive device event notifications. This is now handled by the library and does not need a WndProc override in the user code
  - General code clean-up

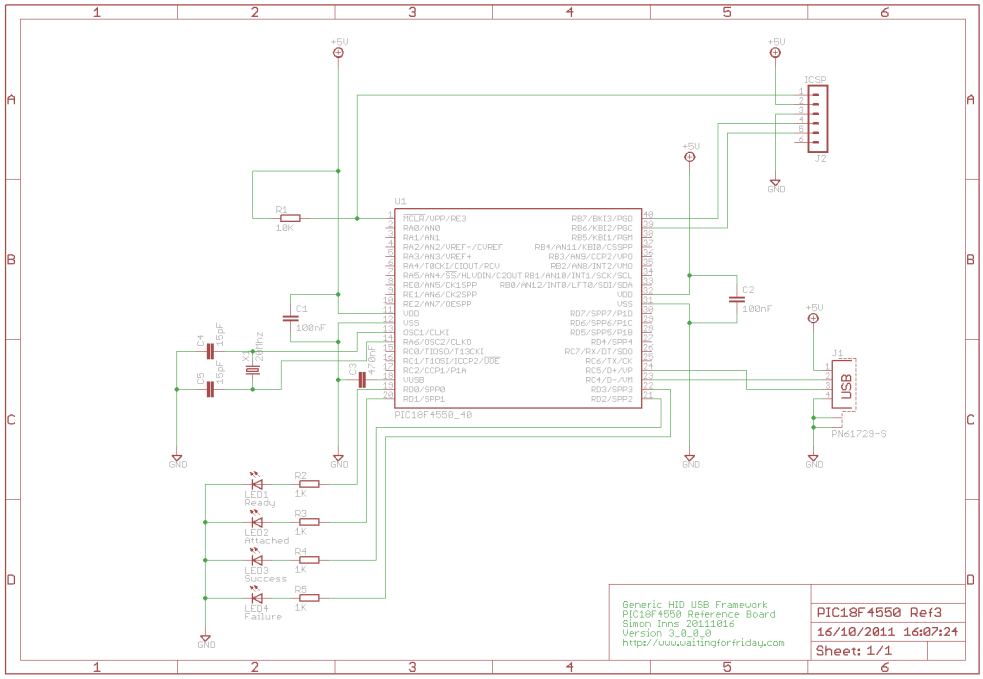
In addition I've also updated all of the circuit diagrams and included a new schematic for the PIC18F2550.

Please ensure that you specify which version of the USB framework you are using when asking for help in the comments.

Reference Hardware

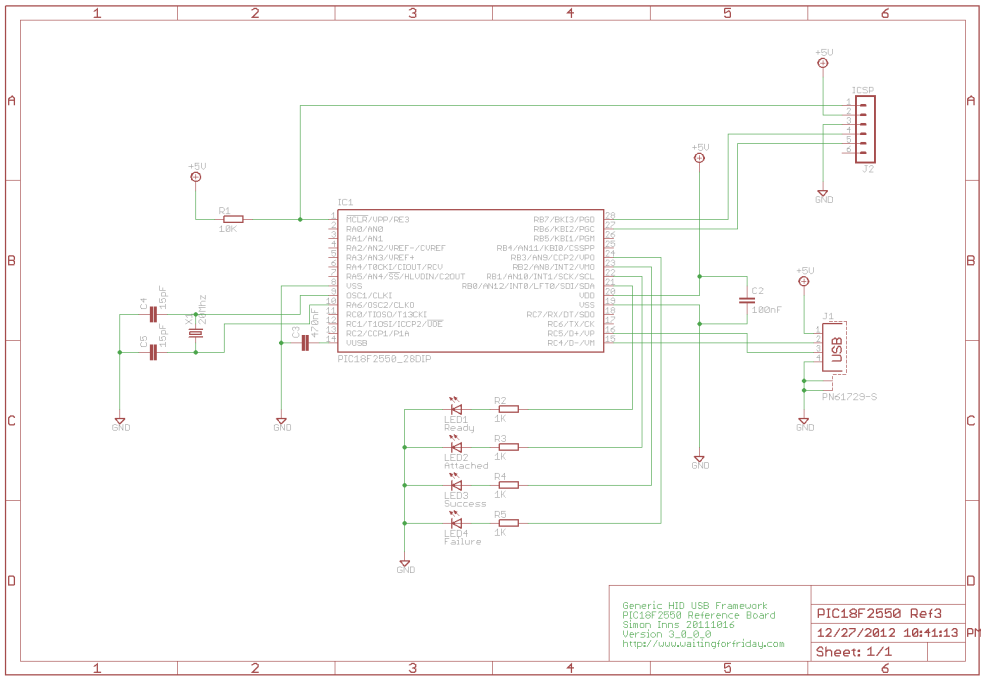
The reference hardware is extremely simple and can be built on a breadboard if required. I've made the pin-out of the hardware identical to the PIC DEM FS USB board available from Microchip to allow you to buy a reference board (if you want to!), you can also use my clone of the Microchip board – the PIC USB Development Board. If you don't have the time/money/inclination to make a reference board here is a circuit schematic of the minimum required hardware for the reference board for both the PIC18F4550 and PIC18F2550 microcontrollers:

PIC18F4550 Circuit Schematic



PIC18F4550 reference schematic

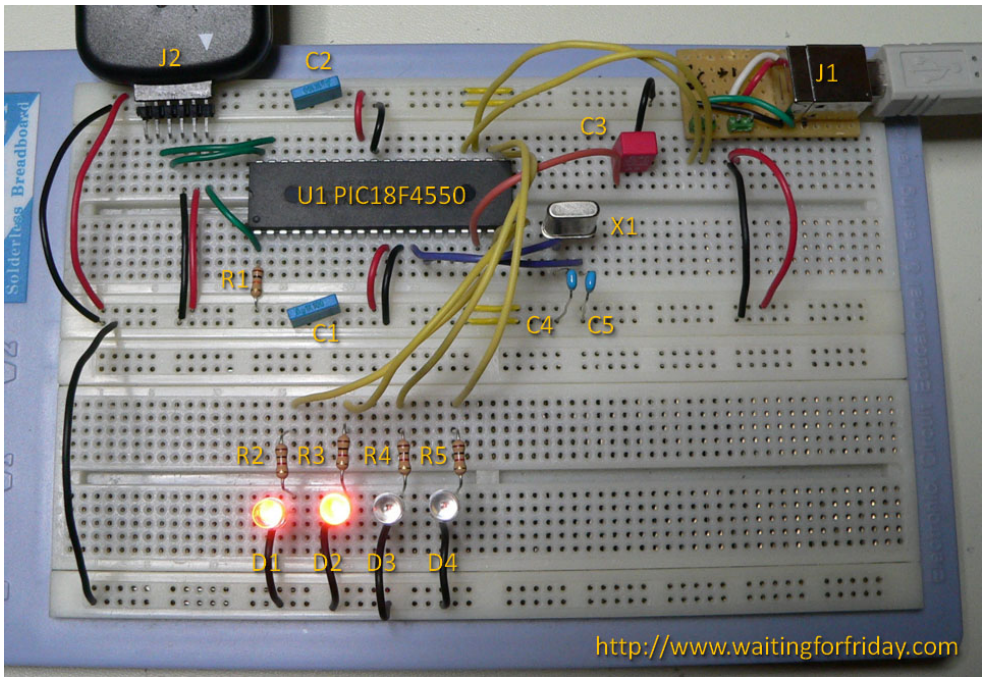
PIC18F2550 Circuit Schematic



PIC18F2550 reference schematic

Breadboard Layout

The next picture shows the reference hardware built on an experimenter's breadboard (using the PIC18F4550):



Reference hardware built on an experimenter's breadboard

The reference hardware provides 4 LEDs which are used to show the status of the USB device as it processes the various requests that can be sent to it from the Windows host-application. The 4 LEDs (from left-to-right) mean:

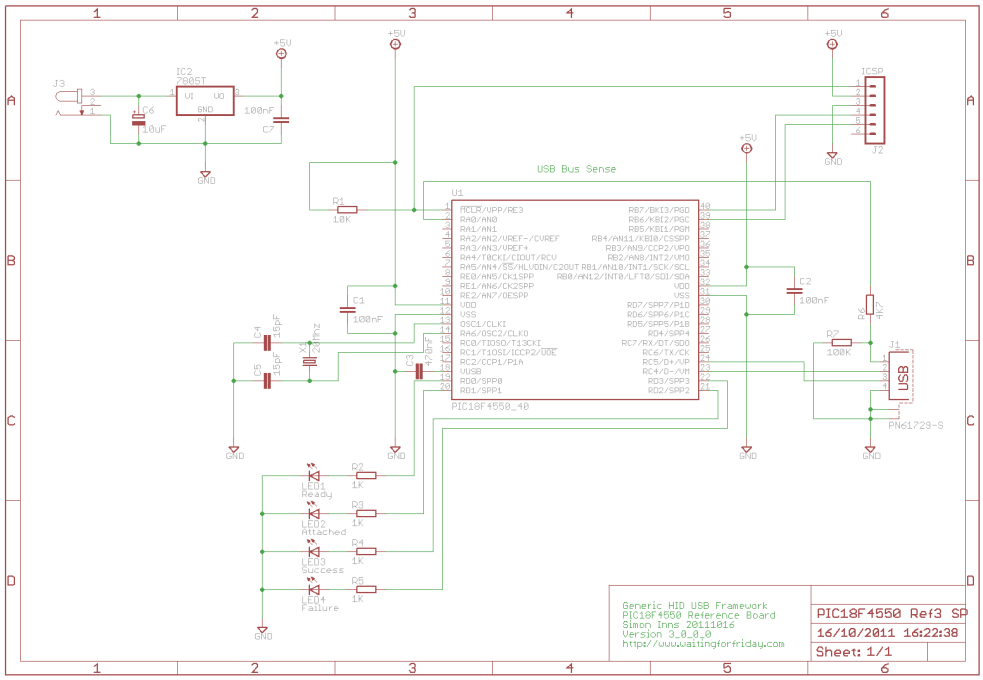
1. Power is on – device is ready
2. USB enumerated – device is ready to communicate
3. Success indicator (this flashes as commands are correctly processed)
4. Failure indicator (this flashes if a command was not correctly processed)

The board provides all you need to add in your own devices and circuity as you require, you can even remove the LEDs, the only part the firmware truly requires is the USB port and its associated hardware.

Self-Powered USB Devices

The reference hardware shown above is for a bus-powered USB device (it takes its power directly from the host computer). This requires the fewest parts and gives you a very simple circuit, however the current draw of the device is limited to 100mA. If you want to build devices that consume more power (to drive more hardware such as relays, motors, arrays of LEDs, etc.) you will need to make the device self-powered.

The following circuit schematic shows a simple self-powered version of the reference board. A LM7805 5V regulator is added to regulate the DC input. The 5V input from the USB bus is directed to RA0 on the PIC18F4550 which allows the firmware to detect when the USB cable is connected and disconnected. You can omit this power-sense feature, however that means that your firmware will need to constantly poll the USB stack to see if a cable is connected.



Self-powered version of the reference board

Reference Firmware

The reference firmware is based on Microchip's USB stack version 2.9a. The firmware provides both the USB stack and a simple structure for building PIC18F USB HID firmware around. The firmware provided performs a number of tests (along with the Windows C# application) which allow quick verification that the firmware and application libraries are functioning correctly.

In the firmware you will find (located in the main.c source file) an implementation for several commands that are mirrored in the reference application:

- 0x80: Single packet write to device – This command receives a single command and data packet from the host (which is 64 bytes long), the packet should contain the command byte (in the first byte of the packet) and then the numbers 1-63 in the remaining bytes. This allows the firmware to confirm that the whole packet was received correctly.
- 0x81: Single packet read/write – This is similar to 0x80, however the firmware generates a return packet (filled with similar data) and passes it back to the host
- 0x82: Single packet write, 128 packet read – This command receives a single packet from the host and, in return, sends back 128 64 byte packets containing data. This command simulates the requirements of bulk data transfer from the device to the host (for applications like data-capture, etc.)
- 0x83: 128 packets write, single packet read – This command receives a command from the host followed by 128 64 byte packets containing data. This is basically the opposite of 0x82 where the host is bulk transferring data to the USB device (used in applications like display drivers, etc.)
- 0x84: Single packet write, time-out on read – This command is a little different from the others. It's used to simulate a failure scenario on the host-side where a command is sent and the host expects a reply, but the firmware does not reply (simulating a bug in the firmware). This really doesn't do much on the PIC device side, but it is useful for testing what the host-application does when the event occurs.

Not only does the reference firmware provide a test environment for the host class library, but it also serves as an example of how to communicate bi-directionally with the host for both simple command and response as well as bulk data transfer.

The reference firmware is GPL'd open-source, so you are free to use it and adapt it to your requirements. Also if you have any suggestions for improvement I'd love to hear from you in the comments.

Installing and compiling the reference firmware

The reference firmware is targeted for the Microchip C18 compiler (and will not work with other PIC compilers without alteration). The firmware also relies on the Microchip USB stack (which is linked in

enabling the code to be easily updated as new versions of the stack are release).

To install the firmware you will need to create a directory in which to place both the library and the firmware. You then have to run the Microchip Application Library installer and instruct it to install into a directory called "Microchip Solutions v2011-07-14" under the directory you created in the last step. The firmware is only tested against version 2.9a of the stack (which is contained in the v2011-07-14 installer) and you can download the library installer [http://ww1.microchip.com/downloads/en/softwarelibrary/microchip-application-libraries-v2011-07-14-windows-installer.exe here].

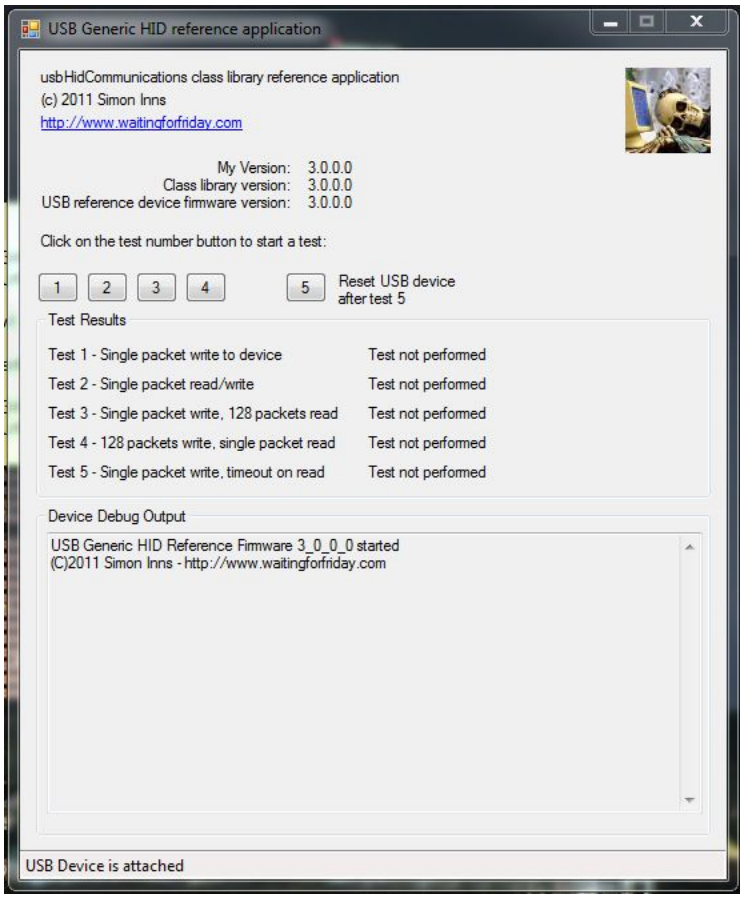
Next grab a copy of the firmware zip file (at the bottom of this article) and unzip it with the application library. If all goes well you should now have two sub-directories in the same directory called "Microchip Solutions v2011-07-14" and "USB_GenericHID_Firmware_3_0_0_0". Now double-click on the "USB_GenericHID_Firmware_3_0_0_0" directory and load the project file USB_GenericHID_Firmware_3_0_0_0.mcp.

If you alter the relative location of either the library or the firmware you will need to change the include path in the firmware project to tell MPLAB where to find the application library. You can do this in MPLAB by going Project -> Build Options Project -> Directories -> Include search path and changing the relative path to match your set-up.

Reference Application

The reference application uses the C# class library to communicate with the reference firmware. It provides a simple interface to the 5 commands available allowing you to select which command you would like to send. The application also creates the commands and data to send to the device as well as interpreting the data and responses sent back by the firmware to confirm that everything worked as it should.

Here you can see the GUI interface from the reference application running on a computer with Windows 7:



Reference Application

The 5 tests correspond to the 5 commands provided by the firmware. When you click on a test button the command is sent to the firmware and both the firmware (through the success and failure LEDs) and the host (providing a 'test passed' or 'test failed' message) provides feedback on the success of the test.

Furthermore, the C# class library provides USB device 'events' to the application letting it know when the USB device is attached and detached from the host. This is tested by removing and plugging in the USB connector whilst watching the status bar at the bottom of the window. If all is well you should see 'device

attached' and 'device detached' messages. Also, the application disables (greys-out) the test buttons when no device is found.

The underlying C# USB generic HID class-library provides a base-class on which to build applications. The reference application defines a class for its own 'specialised' USB device which then inherits all of the accessible functions from the base-class. The application then extends the base-class to include its own methods for communicating with the device. Simply put, the specialised class has to provide an interface from the commands the device must support (0x80, 0x81, etc.) to the 'raw' send and receive methods provided by the class library.

This allows you to be very flexible in how you communicate with the firmware since you are free to define your own commands in any way you like.

Whenever changes, bug-fixed or modifications are performed on the USB generic HID class-library you can use the reference firmware and application to test to ensure that the library is still functioning correctly, this saves a lot of time later on when unforeseen issues pop up (as they always do!).

The USB Generic HID C# Class Library

There are two ways to approach the class-library namely "How do I use it?" and "How does it work?".

If you are interested in the inner workings of the class library I suggest you load it up into Visual Studio and take a look at the code. I've tried to structure it as clearly as possible and also comment it well. Most of the complexity is borrowed, adapted and (in certain cases!) written from scratch. By making the class-library specific to the issues of Generic HID communication with microcontrollers I've managed to remove a lot of the complexity of writing truly generic drivers.

There is also a lot of built-in debugging information. If you compile the reference application in Visual Studio's 'debug' mode you will see a complete log of the application (and class-library's) activities once you exit the application.

Using the class-library

Once you've included the class library in your project (for an example of how to do this simply take a look at the reference application project files which are included below), you will need to add in some lines of C# code to your main form to get the class-library up and running.

First up is the code which needs to go into the class constructor of your main form (usually called Form1()):

```
public Form1()
{
InitializeComponent();

// Create the USB reference device object (passing VID and PID)
theReferenceUsbDevice = new usbReferenceDevice(0x04D8, 0x0080);

// Add a listener for usb events
theReferenceUsbDevice.usbEvent +=
  new usbReferenceDevice.usbEventsHandler(usbEvent_receiver);

// Perform an initial search for the target device
theReferenceUsbDevice.findTargetDevice();
}
```

This code does 3 things:

1. Firstly it initiates the usbReferenceDevice class (which is a specialised form of the libraries base-class) and passes the VID and PID of the target device
2. It adds a 'listener' function to the form which receives the events generated by the USB device (attach and detach events)
3. Finally it performs a first-attempt at discovering the target USB device (the device might already be plugged in, so no notifications will be sent, even though the device is there)

Next up are two more items required by the class-library:

```
        // Create an instance of the USB reference device
        private usbReferenceDevice theReferenceUsbDevice;

        // Listener for USB events
        private void usbEvent_receiver(object o, EventArgs e)
            {
            // Check the status of the USB device and update the form accordingly
            if (theReferenceUsbDevice.isDeviceAttached)
                {
                // Device is currently attached
                }
            else
                {
                // Device is currently unattached
                }
            }
```

The first command generates an 'instance' of the usb device class which is used by the first code example to get the class up and running.

The second method is the USB event listener which is called by the class library when the USB device is detached or attached and allows the application to perform tasks such as updating the status bar and enabling/disabling form controls (this event driven method avoids the usual polling mechanisms most form-applications use to monitor the USB state and is far more efficient).

Once this is done you are free to add whichever windows, buttons and controls your application requires. The primary integration of the class library is deliberately as simple as possible. However to communicate with your USB device there is a little more work to be done.

To represent your own particular USB device you must generate a class which represents the device. This is as simple as adding a class to your project. The class must inherit the base library class and also define some methods to allow your application to communicate with the USB device. Don't worry if this sounds complex, it really isn't, and the reference application provides all the example code you need in the usbReferenceDevice.cs source file.

The first part of this class defines the class itself and creates it's own 'constructor' to allow you to add in any initialisation your own device might need. The code looks like the following:

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Diagnostics;

namespace USB_Generic_HID_reference_application
    {
    using usbGenericHidCommunications;

    class usbReferenceDevice : usbGenericHidCommunication
        {
        // Class constructor - place any initialisation here
        public usbReferenceDevice(int vid, int pid) : base(vid, pid)
            {
            }
```

In the code example you can see the correct way of defining the class (using the class library as the base) and how to 'pass-through' the VID and PID from your constructor to the class library.

Once this is done you are free to add in whatever methods your device requires. Primarily, these will be methods which communicate to and from the device. In the reference application you have 5 example

methods which use the four different communication commands supported by the reference firmware (as well as the 5th command which is purely for demonstrating how the class reacts to firmware bugs).

Here is an example method which sends a command to the USB device:

```
        public bool test1()
            {
            // Test 1 - Send a single write packet to the USB device

            // Declare our output buffer
            Byte[] outputBuffer = new Byte[65];

            // Byte 0 must be set to 0
            outputBuffer[0] = 0;

            // Byte 1 must be set to our command
            outputBuffer[1] = 0x80;

            // Fill the rest of the buffer with known data
            int bufferPointer;
            Byte data = 0;
            for (bufferPointer = 2; bufferPointer < 65; bufferPointer++)
                {
                // We send the numbers 0 to 63 to the device
                outputBuffer[bufferPointer] = data;
                data++;
                }

            // Perform the write command
            bool success;
            success = writeRawReportToDevice(outputBuffer);

            // We can't tell if the device received the data ok, we are
            // only indicating that the write was error free.
            return success;
            }
```

As you can see from the source it's very simple, the class library takes care of all the details. You simply assemble the command and data you wish to send and call the write method, then all that's left it to test the result to ensure it was successful.

There are very few 'public' methods that the class provides, but here is a list of them and what they are used for:

findTargetDevice()

This method searches for the USB device based on the VID and PID which was supplied when the object was created.

handleDeviceNotificationMessages()

This method deals with interpreting the windows notifications received by the application and seeing if they could relate to the USB device. If they do, then the method starts searching for the USB device, or registers it as detached dependent on the messages received.

onUsbEvent()

This method is part of the event mechanism used by the class-library to inform your application of the USB device being attached or detached.

readMultipleReportsFromDevice()

This method is used for incoming bulk data transfers, see the usbReferenceDevice.cs source file (in the reference application) for details of how this is used.

readSingleReportFromDevice()

This method is used to get a single report packet from the USB device. Again, usbReference.cs is the best place for an example of how it's used.

registerForDeviceNotifications()

This method is used to instruct windows to send device notifications to your application (which are then picked up by handleDeviceNotificationMessages() for processing)

writeRawReportToDevice()

This method is used for sending commands and data to the USB device. Again, usbReference.cs is the best place for an example of how it's used.

isDeviceAttached

This property can be tested to see if a USB device is present. It has a value of true if the device is attached and false if the device is detached.

The C# class-library and the reference application are both released under the GPL so you are free to use them and adapt them to your needs. I don't pretend to be an expert in C# (!) so if you have any suggestions for improvements, or bug reports, let me know in the comments!

Conclusions

With a complete reference environment and test suite you have everything you need to create your own advanced USB projects. There is no end to the hardware you can interface to your PC using USB and PIC microcontrollers. Even if you're not a great fan of the PIC, the class-library is generic to all microcontrollers which support Generic HID protocol communication.

As with all learning-experiences, start simple and slow and gradually dig deeper in to the way the class-library operates and how Windows applications are created in C#. There are endless possibilities for your next hack!

Currently the class-library doesn't support more than one USB device with the same VID and PID as this makes the device selection process more difficult (for both the library and the application). I've included some pretty comprehensive device enumeration methods in the class, so adding in support should be straight-forward. I hope to support this soon in an updated version of the library.

Files for download

Visual Studio 2010 C# project files and source for the Reference Application and the Generic HID USB class-library version 3_0_0_0:

USB_Generic_HID_reference_application_3_0_0_0

The MPLAB PIC18F4550 firmware project and source files for Microchip C18:

USB_GenericHID_Firmware_3_0_0_0

The three schematic drawings featured in this article (in Eagle CAD format):

USB_Generic_HID_Schematics_3_0

Example Application – USB LCD Screen Device

To demonstrate the flexibility of the framework I created a simple USB device which allow you to control an LCD display from Windows. The display is a simple HD44780 compatible 16×2 character display. Please see the PC Case USB LCD project page for more details.

See also

Building a PIC18F USB device – Note: this is based on the old C++ library, I hope to update it soon

PIC USB Development Board

PC Case USB LCD – Note: this is based on the version 2 framework, I hope to update it soon

Open Source Visual C++ Class for USB Generic HID Communication – This project is obsolete, the C# library is the only version which is under current development.

← Previous Post Next Post →

7 Responsesso far.

steve248 says:
March 1, 2017 at 6:38 pm

Great article and project, thank you. However, I am having a problem. I am building a keyboard encoder on a pic18f4455 and have got it working. The problem is when I try to get the c# application to communicate with it.

I am successfully editing the PID and VID, the device is recognised by windows and I can see the VID and PID in the device log. However your application will not see a usb device.

I checked that to see whether the software sees other usb devices by changing the VID and PID and it works with any other USB device, the only one that doesnt work is mine.

Is it perhaps because it is a keyboard device? I am not worried about the tests, just getting it to confirm that it can see the device.

Any advice would be greatly appreciated.

Regards

Steve

LOG IN TO REPLY

Simon Inns says:
March 2, 2017 at 6:12 am

Take a look at the C64 VICE front-end project on this site. It implements a USB keyboard using similar firmware. It should act as a good reference for your project.

LOG IN TO REPLY

steve248 says:
March 2, 2017 at 2:12 pm

Thank you, Ill try it.

LOG IN TO REPLY

steve248 says:
March 2, 2017 at 2:22 pm

Ah, I have had a look but my issue is only with the C# application communicating with the device, everything else is working perfectly.

The C64 Vice project doesn't include any software implementation. Am I missing something or did you misunderstand the issue I was having?

Many thanks

Steve

LOG IN TO REPLY

Simon Inns says:
March 3, 2017 at 6:58 pm

USB Generic is only for generic devices. Keyboards and mice don't require a host-side program; they work using the in-built drivers. The Vice project is an example of this; sorry, should have been clearer in my response!

LOG IN TO REPLY

stas0 says:
April 16, 2017 at 6:12 pm

I create project from Microchip MLA usb -> hid_custom -> picdem_fs_usb and when I connect my device to PC it fount my device like. When I use you VS project and change VIP/PID I see that program found device and after several seconds change status to "detach". HIDSharp library connect to device and send messages but it is very simply library and I wanna try library in your project. What is wrong?

LOG IN TO REPLY

Simon Inns says:
April 16, 2017 at 6:45 pm

If it disconnects after attaching then the communication failed (the USB device did not respond to the commands sent by the host). If you are using exactly the code in the project for both the host and the device it may be a bug. If you are using some other firmware on the device, then you need to make it support the right commands (look at the original firmware for details of what's required). Also, turning on debug on the host would probably help you too.

LOG IN TO REPLY

Leave a Reply

You must be logged in to post a comment.