Introduction

USB connectivity is a desirable trait for many embedded devices. Being able to connect to a larger computing system widens the scope of possibilities for such devices. As opposed to legacy connectivity options (such as RS-232) and wireless options (such as Bluetooth), USB enables the simultaneous powering of and data access to such devices. The hardware cost of such an implementation is relatively low but the software needed to enable such interaction is complex.

Fortunately we can stand tall on the works of others. One such developer is Simon Inns, who has developed open source frameworks for the PIC18F and Atmega series microcontrollers. Another is Paul Stoffregen, who has developed the popular modular Teensy 2.0, and who provides accompanying lightweight firmware enabling USB connectivity as well. This note serves only to tie their work together in a complete verifiable solution for quickly enabling USB connectivity for any embedded device, and maintaining UI responsiveness while simultaneously managing USB communication.

Hardware

Any properly laid out circuit board with one of the following Atmel USB-enabled microcontrollers:

Atmega32U4, AT90USB162, AT90USB646, AT90USB1286

This microcontroller should have an external 16 MHz crystal oscillator with matching capacitors and a USB connector. An 8 MHz crystal is acceptable but will require some adjustment later. Internal 8 MHz RC oscillators are not recommended due to their large error range. Atmel RC oscillators are specified anywhere from 7.3 to 8.1 MHz, not good when USB 2.0 requires a PLL at precisely 480 MHz!

Firmware

The only additional firmware needed is the raw HID source files (usb_rawhid.h, usb_rawhid.c, and associated Makefile) available at the PJRC website.[1] Mr. Stoffregen has already written a good amount of guidance regarding how to use the source files on his website and in the source itself, such as how to provide unique identification, specify current draw through the descriptors, etc.

If the hardware has an 8 MHz crystal, these source files will need to be slightly modified. In the C file, the PLLCSR register needs to be set to the value appropriate for an 8 MHz crystal. This depends on the particular microcontroller in use, for the Atmega32U4 (the basis for Teensy 2.0) the value needs to be changed from 0x12 (for 16 MHz) to 0x02 (8 MHz).

```
#elif defined(__AVR_ATmega32U4__)
#define HW_CONFIG() (UHWCON = 0x01)
#define PLL_CONFIG() (PLLCSR = 0x12) // 0x12 for 16 MHz, 0x02 for 8 MHz
#define USB_CONFIG() (USBCON = ((1<<USBE)|(1<<OTGPADE)))
#define USB_FREEZE() (USBCON = ((1<<USBE)|(1<<FRZCLK)))
```

[1] http://www.pjrc.com/teensy/rawhid.html

Also in the Makefile, the definition for F_CPU needs to change from 16000000 to 8000000.

```
# Processor frequency.
#     This will define a symbol, F_CPU, in all source code files equal to the
#     processor frequency. You can then use this symbol in your source code to
#     calculate timings. Do NOT tack on a 'UL' at the end, this will be done
#     automatically to create a 32-bit value in your source code.
F_CPU = 16000000
```

A skeleton implementation of the Teensy firmware looks like this. Checking for received data is done continually in a while loop. (This may seem simplistic, but you can muck about with the logic to achieve more sophisticated behavior.) The receive function returns a value of 1 only if data has been received and placed in the 64-byte receive buffer. A timeout value in milliseconds can be specified in place of 0.

```
#include "usb_rawhid.h"
#define CPU_PRESCALE(n)     (CLKPR = 0x80, CLKPR = (n))
uint8_t rxBuffer[64]; // USB receive buffer
uint8_t txBuffer[64]; // USB transmit buffer

int main(void)
{
	setupHardware();     // do what you gotta do
	int r;
	CPU_PRESCALE(0);     // makefile sets the clock freq

	usb_init();
	while (!usb_configured());

	// Wait an extra second for slower PCs
	_delay_ms(1000);

	while (1)
	{
		// check for received data
		r = usb_rawhid_recv(rxBuffer, 0);
		if (r > 0)
		{
			// do something with it

			// send a response
			usb_rawhid_send(txBuffer, 3);
		}
	}
}
```

There is a limit to how fast communication can occur between a PC and a device running this firmware. I believe it varies based on the computer in question, but the minimum reliable transmit timeout (in the send function) I have found to be reliable is 3 milliseconds. Any lower and packets end up lost. Part of this is due to limitations in the HID spec; it is not intended for high data throughput and engineers designing for lots of real-time data transfer should be aware of this.[2]

[2] http://www.waitingforfriday.com/forum/viewtopic.php?f=18&t=3167

Software

The required PC source code is available in a C# package on Mr. Inns' website.[3] (Accompanying reference firmware based on LUFA is also available here but has a larger memory footprint. Interested users should check Dean Camera's LUFA stack[4] which is more versatile but overkill for our needs here.) Mr. Inns has gone into sufficient detail on how to use his library for HID communications; relevant user code has been provided and commented on here.

```
public mainForm()
{
    InitializeComponent();
    // Create the USB reference device object (passing VID and PID)
    Device = new usbReferenceDevice(0x00B4, 0x01B0);
    // Add a listener for usb events
    Device.usbEvent += new usbReferenceDevice.usbEventsHandler(usbEvent_receiver);
    // Perform an initial search for the target USB device (in case
    // it is already connected as we will not get an event for it)
    Device.findTargetDevice();
}
```

Add the USB device initialization to the main form initialize routine; simply declare a new USB device as a member of the Form. Call its functions (like the W1R1 below) like you would any member of any class.

```
public bool W1R1(Byte[] outputBuffer, ref Byte[] inputBuffer)
{
    // Perform the write command
    bool success;
    success = writeSingleReportToDevice(outputBuffer); // buffer length must equal 64
    // Only proceed if the write was successful
    if (success)
    {
        // Perform the read
        success = readSingleReportFromDevice(ref inputBuffer); // buffer length 64
        // Alternate function for multiple packet reading (buffer length = 64*128)
        success = readMultipleReportsFromDevice(ref inputBuffer, 128);
    }
    return success;
}
```

The important functions are the `writeSingleReportToDevice` and the `readSingleReportFromDevice` functions. There is also a `writeMultipleReportsToDevice` function but I am having difficulty imagining an application requiring its use. (Handling such a function in the actual device would require some modification to our skeleton firmware above.) These return a Boolean indicating success or failure. Error handling should be done back in the main form calling this function. Also Mr. Inns' source constructs the buffers inside the usbReferenceDevice class; I prefer to construct them outside.

[3] http://www.waitingforfriday.com/index.php/USB_Generic_HID_Open_Source_Framework_for_Atmel_AVR_and_Windows
[4] http://www.fourwalledcubicle.com/LUFA.php

Maintaining a Responsive UI

This basic setup works just fine for the majority of HID applications. However there are times where waiting for a USB transaction to complete is undesirable and continuous communication and updating become necessary. Fortunately C# makes it easy to implement multithreading, allowing the UI to be managed by one thread and USB communication to be managed by a background thread. The background thread updates the main thread with the data it receives, and the main thread handles any other user input, maintaining the desired responsiveness of the PC application.

There are many good guides on the theory of multithread coding that are freely available. I personally found the e-book by Joe Albahari to be easy to understand.[5] Adapting multithreading to Mr. Inns' USB HID source code is relatively simple. The background worker thread in this case needs to construct or receive an output buffer, and handle received data. Based on the application, it may need to run continuously, and be cancellable by the PC user, by the device itself, or both.

```
Byte[] buildOutput();
double update(Byte[]);

void AskWorker_DoWork(object sender, DoWorkEventArgs e)
{
    bool success = false; double Vupdate = 0;
    Byte[] inputBuffer = new Byte[64];
    Byte[] outputBuffer = new Byte[64];
    outputBuffer = buildOutput();
    do
    {
        success = Device.W1(outputBuffer);
        if (!success) { e.Cancel = false; return; } // quit on device fail
        success = Device.R1(ref inputBuffer);
        if (!success) { e.Cancel = false; return; } // quit on device fail

        // this updates the main form with the received data
        Vupdate = update(inputBuffer);
        Action action = () => Vc.Text = Vupdate.ToString("F0");
        this.Invoke(action);

        Thread.Sleep(100); // don't spam the device! Adjust as needed
    }
    while (!((BackgroundWorker)sender).CancellationPending); // quit on user cancel
    e.Cancel = true; return;
}
```

In the background worker thread the USB functions have been separated into write only and read only functions (the W1 and R1 functions, respectively). This prevents hang-ups during the read operation if the write operation fails (for example if a device is disconnected). A main form update function follows. The condition for exiting the loop is if the user cancels (a button must be provided on the main form for this) or if USB communication fails.

[5] http://www.albahari.com/threading/

Upon exiting the `DoWork` routine the `RunWorkerCompleted` function is automatically executed. This only needs to be a simple function, checking the status of e.Cancel and/or e.Result, and exiting appropriately. For most background worker threads of this type, a silent exit is best.

```
void AskWorker_RunWorkerCompleted(object sender, RunWorkerCompletedEventArgs e)
{
    if (e.Cancelled)
        // exit quietly
    else
        MessageBox.Show("Communication error, recommend power cycle.");
}
```

Finally the background worker needs to actually be initiated. Add a BackgroundWorker to the form:

```
BackgroundWorker AskWorker = new BackgroundWorker();
```

And initialize it along with everything else:

```
public mainForm()
{
    InitializeComponent();
    Device = new usbReferenceDevice(0x00B4, 0x01B0);
    Device.usbEvent += new usbReferenceDevice.usbEventsHandler(usbEvent_receiver);
    Device.findTargetDevice();

    AskWorker.WorkerSupportsCancellation = true;
    AskWorker.DoWork += AskWorker_DoWork;
    AskWorker.RunWorkerCompleted += AskWorker_RunWorkerCompleted;
}
```