

Asynchronous PIC-to-PC USB Interrupt Transfers

Contributed by Evan
 Saturday, 04 November 2006
 Last Updated Tuesday, 06 February 2007

Just a disclaimer, when I use the term "asynchronous" here I am talking about from the perspective of the main program, with the actual workings of the USB communication between the PIC and the USB host controller viewed as abstract.

This is an example of sending packets from the PIC to the PC asynchronously. The code in my 18F4550 USB Interfacing article detailed only how to perform command/response communications where the PC sends a command, and the PIC immediately sends a response, and thus if you wanted the PIC to be able to send commands to the PC at any given time, you would have to poll it continuously from the PC, wasting a lot of bandwidth repeatedly sending packets back and forth. That kind of behavior is especially inefficient when you consider what is happening on the actual USB communication level with the PIC set up for interrupt transfers: the host controller is ALREADY polling the PIC every so often to see if it has any packets to send, and thus any explicit polling we do on top of that is completely redundant.

Of course, allowing data to be received from the PIC asynchronously is a bit more challenging. On the PIC side it's not too bad; if you choose to use the same endpoint that you use for the other command/response type communications, then it's trivial. I chose to add an additional endpoint (EP2) and dedicate it to these upstream interrupt transfers, so as not to interfere with any other communications. This requires some work in various places; descriptors, #defines, etc. You can see more about what this takes in my Understanding the Microchip USB Firmware article. After that, it's really just a matter of cloning a couple of things from the USBGen code, and that's it.

On the PC side, the main change is that when opening a pipe to that endpoint, you have to open it with "_ASYNC" at the end, ie - "\\MCHP_EP2_ASYNC" for endpoint 2. The result is that incoming data is placed in a buffer of length 100. This change is documented in the comments of the C++ source code for mpusbapi.dll. To read data from the buffer, you simply call MPUSBReadInt() instead of MPUSBRead(). So, in your code you still have to poll on some level, however there's no extra USB traffic being generated. The logical progression now is to stick the polling of the buffer in a separate execution thread, and simply have it send data back to the main thread when anything is received, which is what I have done in this example program.

To take it a step further (making it very easy to add this feature to any program, with just a few lines of code), I set everything up such that when a new packet is received, an event is fired, and the packet is included in the event args. The class PICAsyncManager handles just about everything you need; in your main form/code/whatever, you create an instance of it, add it as a control, tell it to start reading, and then register a function of your choosing to handle the event. The entirety of the relevant code for the form in the demo application is this:

```
public partial class Form1 : Form
{
    PICAsyncManager pam;
    public Form1()
    {
        InitializeComponent();
        pam = new PICAsyncManager();
        this.Controls.Add(pam);
    }

    private void button1_Click(object sender, EventArgs e)
    {
        if (!pam.IsRunning)
        {
            pam.Start();
            pam.EventNewUSBIntData += new NewUSBIntDataEventHandler(AppendNewUSBIntData);
            textBox1.AppendText("Interrupt monitor thread started...");
        }
        else
        {
            pam.Stop();
            textBox1.AppendText("Interrupt monitor thread stopped...");
        }
    }

    private void AppendNewUSBIntData(NewUSBIntDataEventArgs e)
    {
```

```
        textBox1.AppendText(e.newdata[1].ToString()+" ");  
    }  
}
```

So what does this example do? The PIC simply repeatedly sends a byte, increments it, and repeats. On the PC side, you click a button to start listening for transfers, and the incoming values get sent back the main thread and appended into the text box. Thus, the end result is that you see a continuous up-count in the text box. [{nomultithumb} DOWNLOAD THE DEMO CODE](#)

(last updated 2-6-07) Note: if you have trouble compiling the firmware, and get an error about the path length exceeding 63 characters, go into Project>Build Options>Project, go to the MPLINK tab, and check "Suppress COD-file generation". [{mos_sb_discuss:8}](#)