PIC-USB Debugger Documentation

1.13.2010

Boris Taratutin

Background

When I began working with microcontrollers, I instantly fell in love with them. At the same time, they were a great source of frustration - as, unlike in regular programming, it was extremely difficult to find out what was going on inside of the PIC; once you loaded the code onto the microcontroller, you couldn't access your variables or know if any error flags had been raised, etc. The closest thing to debugging was lighting up LEDs or making them blink in a certain pattern

After learning that the PIC could interface with the USB, I saw the potential for constructing a custom debugger. Of course, proprietary debuggers can be purchased from 3rd party providers, but they are often restrictive, non-customizable/expandable, and cost money. Thus, building a custom one made sense.

I hope it is something that proves useful and takes some of the frustration out of the process of working with PIC microcontrollers. Feel free to add, modify, and distribute this code at will and contact me with suggestions, feedback, and any gratitude you may have.

Cheers!

Current Features

- Live-readout of PIC PORT & TRIS registers
- Ability to send up to 8 messages from the PIC to the USB
- 2 methods of sending data back to the PIC
- Data logging
- Variable data acquisition rate
- Live modification of PORT & TRIS registers

Future Features

- Utilization of high-speed USB
- Simplification and streamlining of GUI code
- More modular data logging, variable readout, etc.
- Vendor request consolidation between python GUI & PIC C code
- Functionality for Snarglemumplimps
- Compatibility with windows 7

- (your ideas here)

Installation

I'm not going to lie, this is the annoying part.

Python

- Download & Install Python 2.6: http://www.python.org/download/
- Download & Install PyQt: http://www.riverbankcomputing.co.uk/software/pyqt/download

USB

Next, we need to register our PIC as a USB peripheral. To do so, open up the project in the main folder and program it onto the PIC (it has all of the necessary code to do some basic testing and associate it as a USB device).
Next, make sure the PIC has the proper connections to a standard USB 2.0 cable, and hook it up to a USB port on your computer. You should hear an annoying 'bling' sound and the 'add new hardware wizard' should pop up.
In response to the wizard, check 'not this time' on the first panel.
On the second panel, specify that you want to select a directory to search for the proper driver in. Then, navigate to USB > LibUSB-Win32 > bin folder (which should contain device-driver.inf).
You will get a warning that the driver is not 'digitally signed'. Disregard and continue.
If everything went well, now every time you plug a microcontroller in that has the appropriate USB code, it will be recognized as a USB device!
Now, you can navigate to the 'python' folder and run the Pic_Gui_Debugger.py file.
These instructions were obtained courtesy of Bradley Minch, on http://ece.olin.edu/poe/handouts.html > Handout 4.1 (pdf)

General Structure of Program

Initialize everything

- Create USB Connection
- Create Menu, status bar, main frame
- Create widgets, link them to actions, organize via layouts, add to display

Define widget actions

- Define a method for each widget action, and do the actual communication

Other methods

- Helper methods for things like data logging, etc.

How it works

A threaded loop runs that constantly calls "get_pic_status" and "update_gui". The first method uses the USB protocol to communicate with the PIC and obtain information about its PORT and TRIS registers, as well as all of the debug messages, etc. By default, this contacts the PIC 20 times a second. This information is then displayed onscreen thanks to a GUI constructed with PyQt. The user can then change values, send messages back to the PIC, and do things like log data.

Understanding *Getting & Sending* Data - Python Side

Getting and sending data between the USB and PIC all depends on one magical line of code. Walking through an example and understanding the parameters is the best way to gain familiarity with the protocol, so that's what we'll do next.

```
def get_led_clicked(self):
        self.usb.control_transfer(self.dev, 0xC0, GET_LED, 0, 0, 1, self.buffer)
        self.led_status = ord(self.buffer[0])
        self.update_gui()
        print "Got LED value of %d" % self.led_status
```

In English: "when the *get_led* button is clicked, this method is called. Get information from the PIC via the USB protocol. Convert received value to an integer. Update the GUI. Print out results."

The dark orange line (self.usb.control...) is the important one. Let's compare it side-by-side to the line for sending data to the PIC:

```
self.usb.control_transfer(self.dev, 0xC0, GET_LED, 0, 0, 1, self.buffer)
```

```
self.usb.control_transfer(self.dev, 0x40, SET_LED, self.led_status, 0, 0, self.buffer)
```

The first line *gets* the value of the LED, and the second line *sends* the value (to the PIC). As you can see, there are 7 arguments, so let's go through them one by one:

1. `self.dev` - This just references the USB device that was instantiated in init_usb(). Not important.

2. `0xC0 / 0x40` - This is easier to think about in binary. So let's convert;

    ```
    - 0xC0: 0b11000000
    - 0x40: 0b01000000
    ```

    As you can see, they are very similar, and do almost the same thing. Without going too deep into it, 0xC0 indicates an "input" or "data request" from the PIC (meaning that it expects to get a value from the PIC). 0x40, however, indicates that the USB is sending data to the PIC

3. `GET_LED / SET_LED` - Remember those arbitrary defined numbers at the beginning of the file? These are references to them. They essentially tell the PIC *what* the command is; what do you want it to do? An identical set can be found/needs to exist in the code that's on the PIC. Thus, when this value gets sent to the PIC, it can be parsed in a case/switch statement, and the appropriate actions carried out. So essentially, this is 'what function on the PIC do you want to call?'

4. `0/self.led_status` - This is the single byte of data that the USB can send to the PIC. Obviously, if

we're requesting data (as in the 'get_led' example), we're not sending any data, and so the value is 0. However, if we want to send a value, (as in the 'set_led' example), then the value we want to send to the USB goes here.

`5. 0/0` - (I'm not totally sure about this one), but I believe if high-speed USB is enabled, this allows you to send a second value to the PIC via the USB.

`6. 1/0` - This is how many values you're expecting back from the PIC. For the 'get_led' function, we want one value back; the LED's status. Obviously, when we're sending values, we're not expecting any values back, so the value is '0'. The maximum value for this is 8.

`7. self.buffer` - the 8-byte buffer that is used for communication. What's important here is that if the USB requests values (up to 8) from the PIC, they will be stored in this buffer. That's why you see the line `ord(self.buffer[0])`, which accesses the first value of the buffer, and the 'ord' converts it into an integer. That's how we get a value back. (if you're requesting 8 values back, you can access them by calls like 'self.buffer[7]', etc.

However, the python GUI is only half of the equation when it comes to getting/sending values between the PIC and the USB. If you want to understand the other half, check out the bottom part of "main.c", entitled 'vendor_requests'. It should explain things clearly enough.

Adding GUI Items in PyQt

(In case you're curious, or want to add/remove things from the GUI)

There are 4 steps to add a GUI widget item in PyQt:

1. Create the widget item
2. If it needs an action, create that action, and connect it to the widget
3. Add the widget to a layout
4. Add the layout to the central widget layout

Example:

Create a pushbutton:

```
self.get_led = QtGui.QPushButton("Get LED status")
```

Connect the pushbutton to an action (ie. on click, call *self.get_led_clicked*):

```
QtCore.QObject.connect(self.get_led, QtCore.SIGNAL('clicked()'),
self.get_led_clicked)
```

Create a horizontal layout, and add the button to that layout:

```
        h_box = QtGui.QHBoxLayout()
        h_box.addWidget(self.get_led)
```

Create the central widget, and add the layout to that widget:

```
        self.central_widget = QtGui.QWidget()
        self.central_widget.setLayout(h_box)
```

Ending Remarks

I hope this was helpful!

- Boris Taratutin (boris.taratutin@gmail.com)