PSoC 5LP “Vendor-Specific” USBFS Tutorial

Eric Ponce

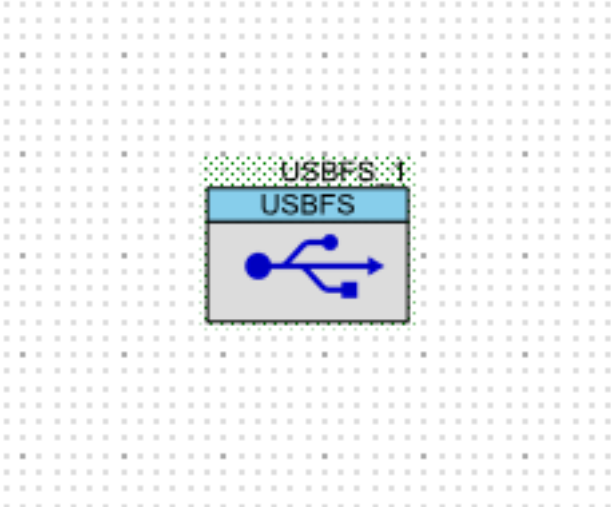
May 9, 2016

Introduction

This tutorial sets up a simple USBFS (USB Full Speed) implementation to echo back sent data on the PSoC 5LP. This example uses Python to interface with the PSoC.

PSoC Setup

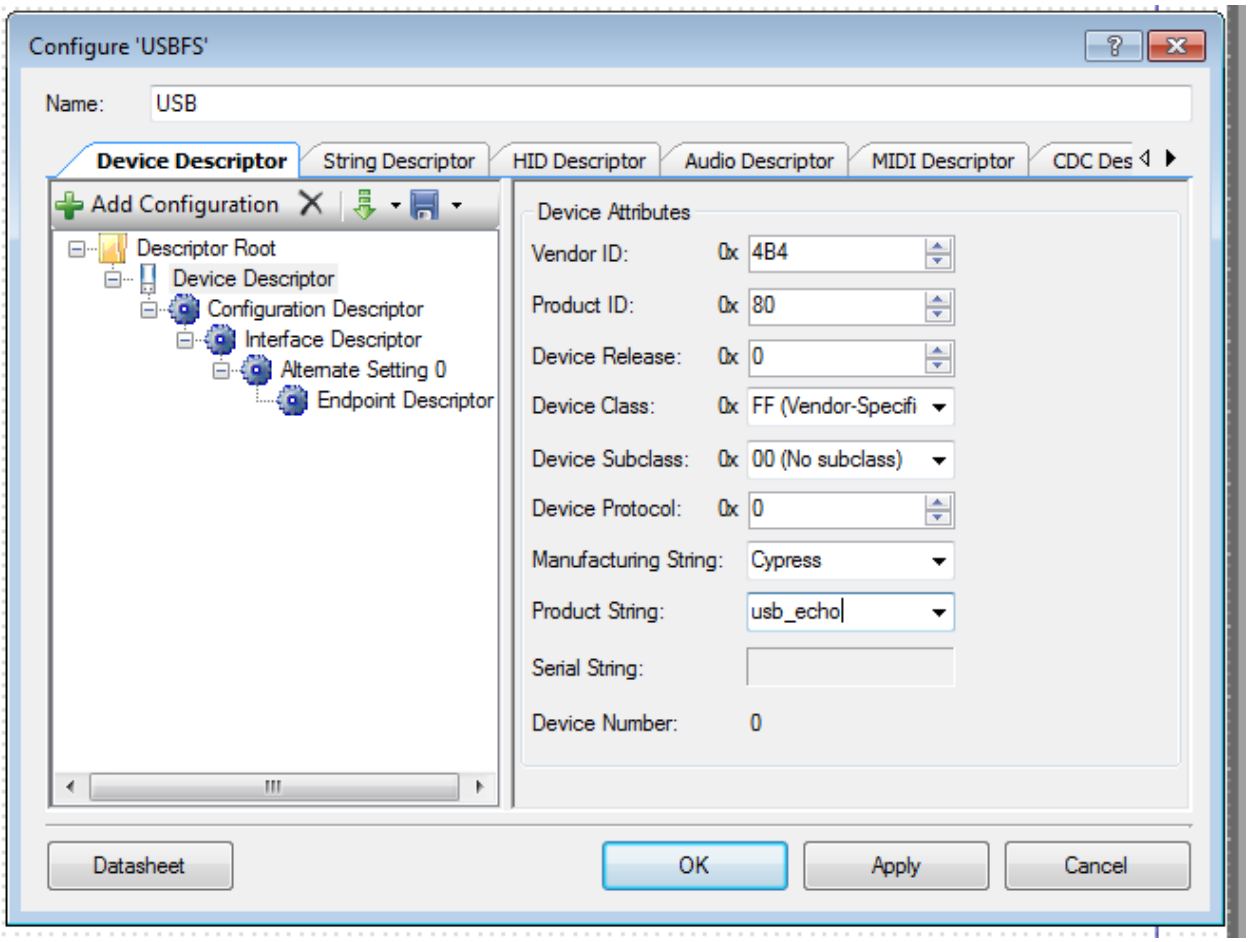
Firstly, create a new project in PSoC Creator for the PSoC 5LP. The supplied example code is called “usb_echo.” In the ‘TopDesign.cysch’ file, drag in a USBFS block under Communications-USB.



Open the configuration screen by double clicking the component. Change its name to USB. Under ‘Descriptor Root’, highlight the ‘Device Descriptor’ label and set the values as follows:

- Vendor ID: 0x4B4
    - This ID describes the vendor for the USB device. In this case, we’ll borrow Cypress’s. Vendor ID’s are assigned by the USB-IF and cost several thousand dollars a year!
- Product ID: 0x80
    - This describes individual vendor products. 0x80 was arbitrarily chosen.
- Device Release: 0x00
    - Version number. Useful for determining product version when designing USB application and potentially distributing firmware updates

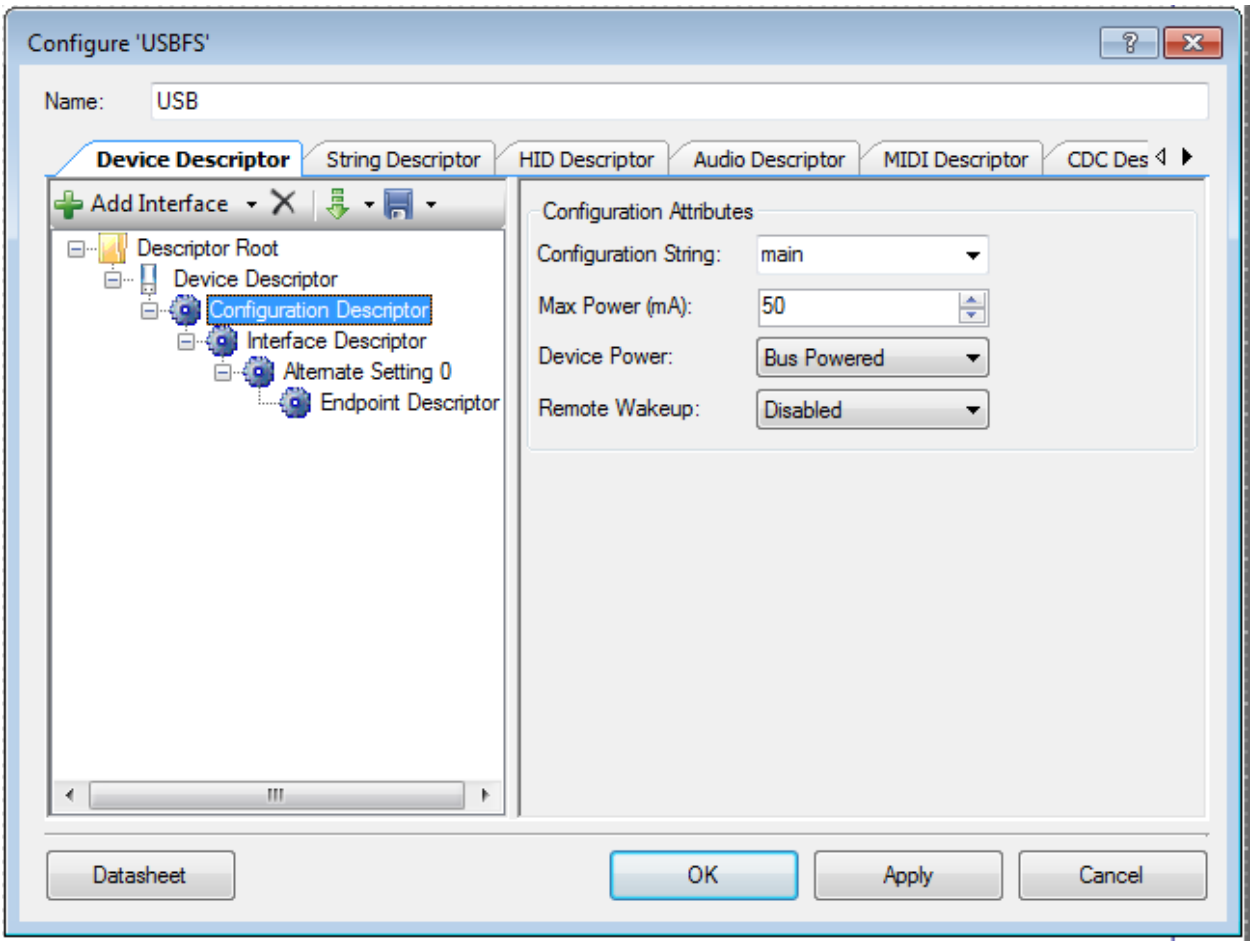
- Device Class: 0xFF (Vendor Specific)
    - Our device does not describe to any of the USB standard device classes so we choose the generic 'Vendor Specific' class.
- Device Subclass: 0x00 (No subclass)
    - We aren't taking advantage of subclasses in this application
- Device Protocol: 0x0
    - Since we're vendor specific, this is also arbitrary.
- Manufacturer String: "Cypress"
    - The manufacturer of the PSoC!
- Product String: "usb_echo"
    - Our product's name



Now, click Configuration Descriptor and set the values as follows:

- Configuration String: 'main'
    - The description of our only configuration. In this case, we called it main
- Max Power (mA): 50mA

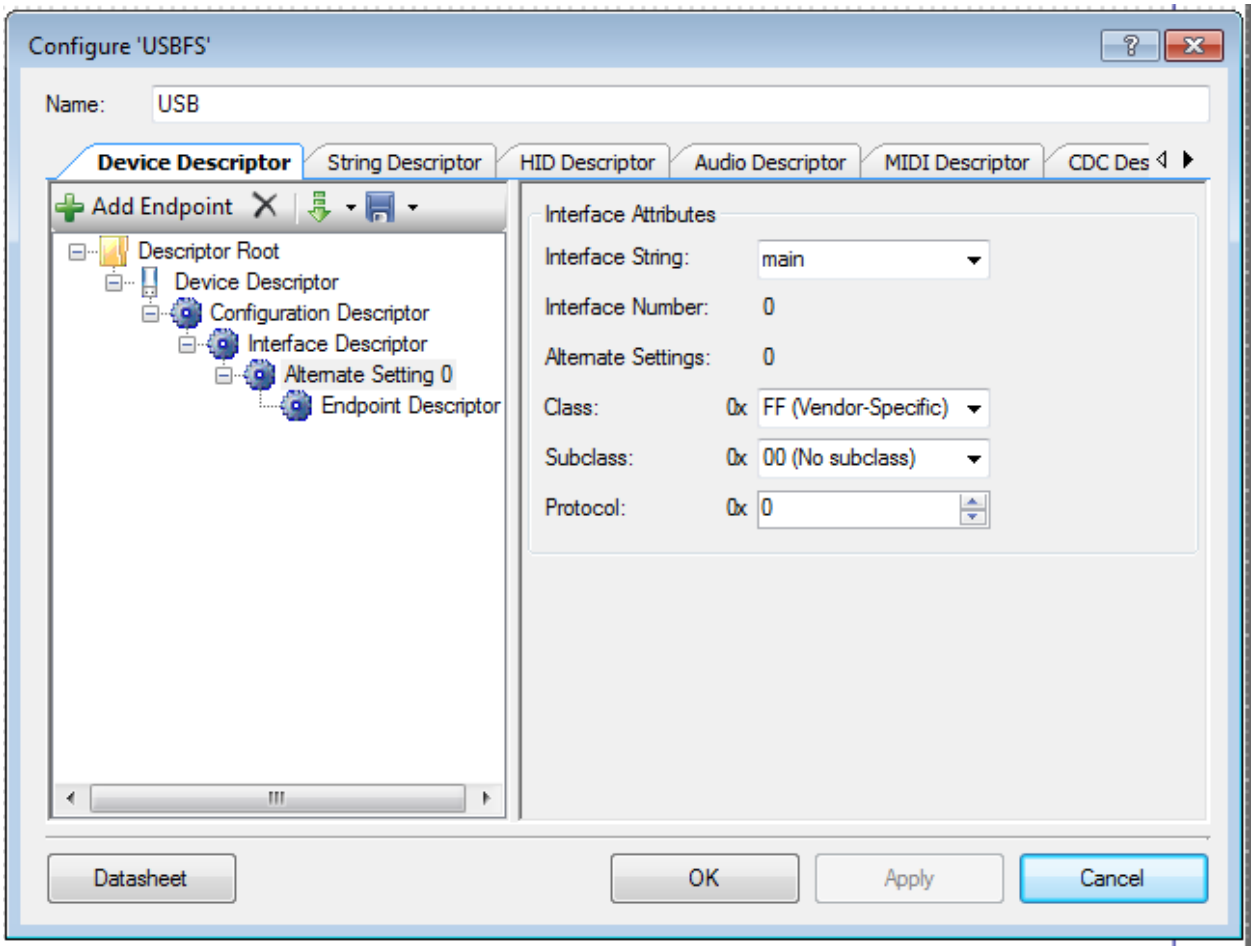
    - This value tells the host how much power the device expects to draw. This allows the host to cut off the device in the case of hardware failures resulting in large current draw.
- Device Power: Bus Powered
    - Tell the host to give us power
- Remote Wakeup: Disabled
    - This application won't use remote wakeup capabilities



Onto 'Alternate Setting 0'. Set the values as shown below:

- Interface String: 'main'
    - Our only interface, so we'll call it main
- Class: 0xFF (Vendor-Specific)
    - Once again, this application doesn't conform to any standard classes, so we'll go with 'vendor-specific'
- Subclass: 0x00 (No subclass)
    - No need for subclasses here
- Protocol: 0x0

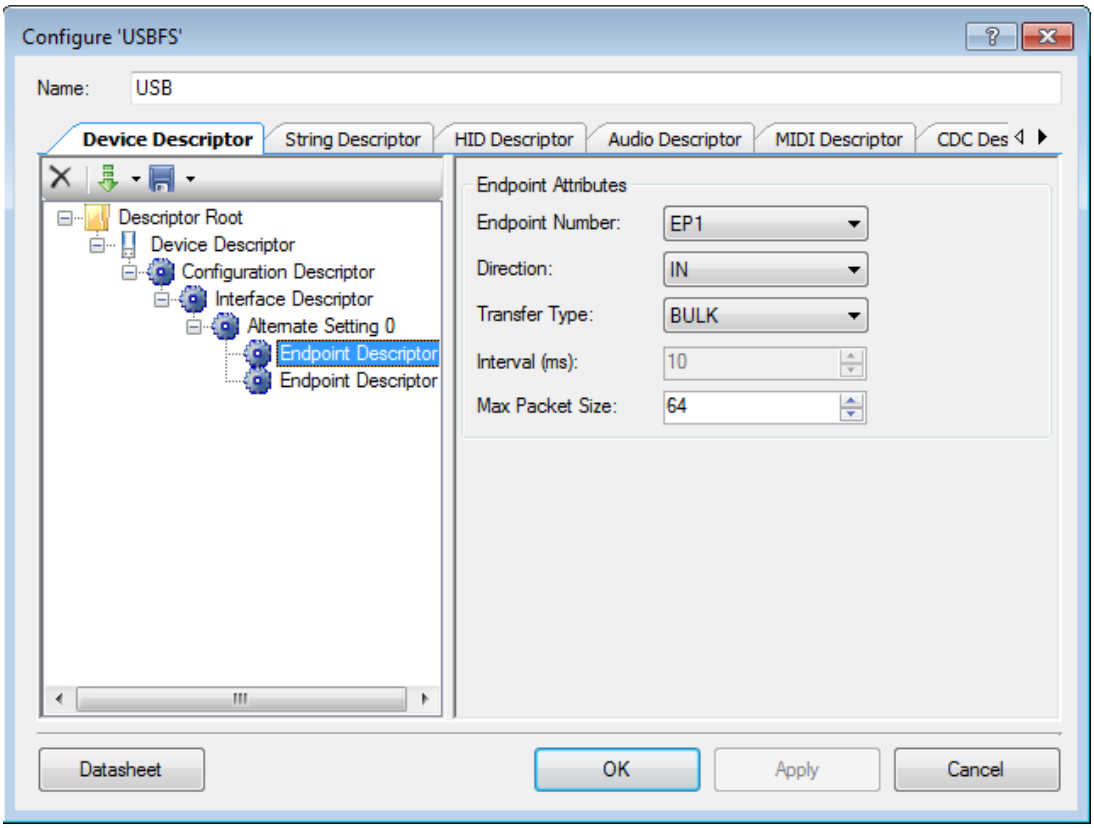
- Arbitrarily decided since we're vendor specific



Here comes the point where we actually decide how we are going to communicate. Endpoints are unidirectional (except for control transfers) communication channels that can be designated to specific tasks by the driver. In this case we need two endpoints, one for in (into the Host or out from the PSoC), and one for out (out from the Host or into the PSoC), so press the "Add Endpoint" button. Select the first endpoint and set the settings as follows:

- Endpoint Number: EP1
  - Each interface can have up to 9 endpoints, but endpoint 0 is reserved for control type transfers. Our first endpoint will be endpoint 1.
- Direction: In
  - This endpoint will be our means of sending data to the Host PC.
- Transfer Type: Bulk
  - There are four transfer types available to endpoints in the USB protocol: Control, Interrupt, Bulk, and Isochronous.
    - Control transfers allow us to perform a variety of command and status operations, such as retrieving all those settings we just set up!

        - Interrupt transfers are useful for applications with non-periodic data transfers requiring quick attention, much like the microcontroller interrupts you should be familiar with
        - Bulk transfers allow for large bursts of data and has guaranteed delivery and error detection
        - Isochronous transfers involve bounded latency data transfers with no delivery guarantees. Useful with real time data when you don't mind losing a few packets.
- Max Packet Size: 64
    - The maximum size of a packet allowable on this endpoint

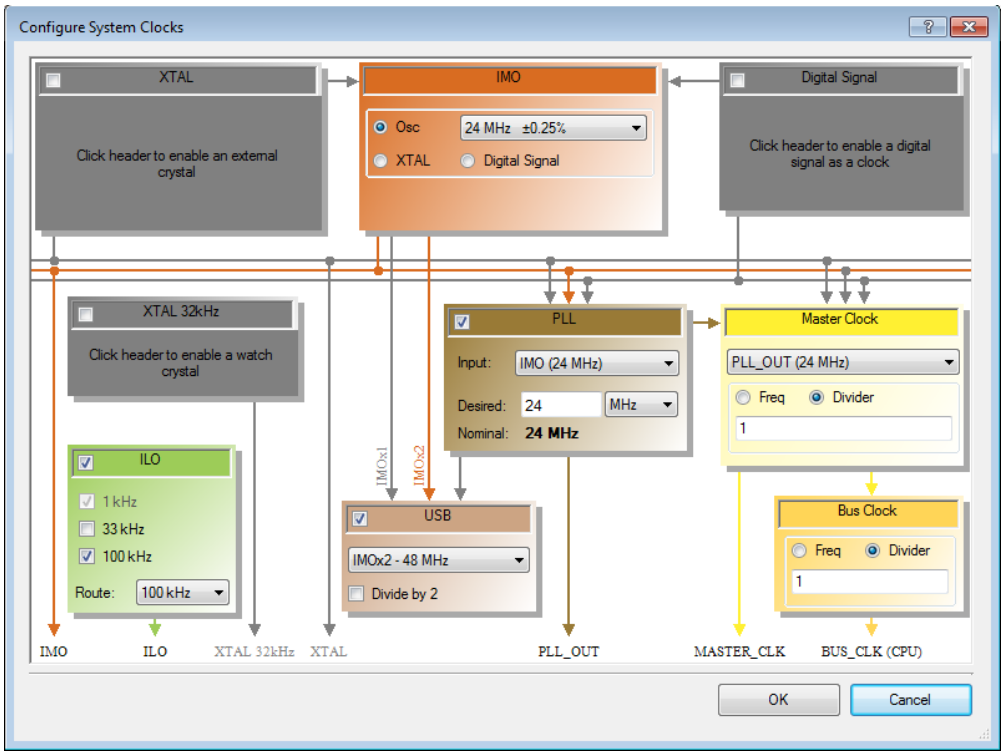


For endpoint 2, set the following values:

- Endpoint Number: EP2
- Direction: Out
- Transfer Type: Bulk
- Max Packet Size: 64

Press okay to save the values and open the design-wide resources file. Navigate to the 'Clocks' tab. Here we will set the clocks to the values required for the tight timing tolerances of USB. Double click the USB_CLK line to open the clock configuration window and set the values described below:

- Set the IMO (internal main oscillator) frequency to 24 Mhz +-0.25%
- Activate 100KHz in the ILO (internal low-frequency oscillator) and select 100KHz to be routed
- Enable the USB clock



Back in the ‘Pins’ tab, we can leave the USB:Dm and the USB:Dp pins empty so that Creator can select the correct pins for us. Compile the program so that Creator generates the necessary APIs for you and then open ‘main.c’. The following code initializes the USB device (and particularly the OUT endpoint) and echoes out and received data. Remember that the OUT endpoint is OUT from the HOST and into the PSoC

```
#include <project.h>

uint8 buffer[512], length;

int main()
{
   CyGlobalIntEnable; // Enable interrupts
   USB_Start(0, USB_3V_OPERATION); // Start the USB peripheral
   while(!USB_GetConfiguration()); // Wait until USB is configured
   USB_EnableOutEP(2); // Enable our output endpoint (EP2)
   for(;;){
      while(USB_GetEPState(2) == USB_OUT_BUFFER_EMPTY); // Wait until we have
data
      length = USB_GetEPCount(2); // Get the length of received data
      USB_ReadOutEP(2, buffer, length);    // Get the data
      while(USB_GetEPState(1) != USB_IN_BUFFER_EMPTY); // Wait until our IN
EP is empty
      USB_LoadInEP(1, buffer, length); // Echo the data back into the buffer
   }
}
```

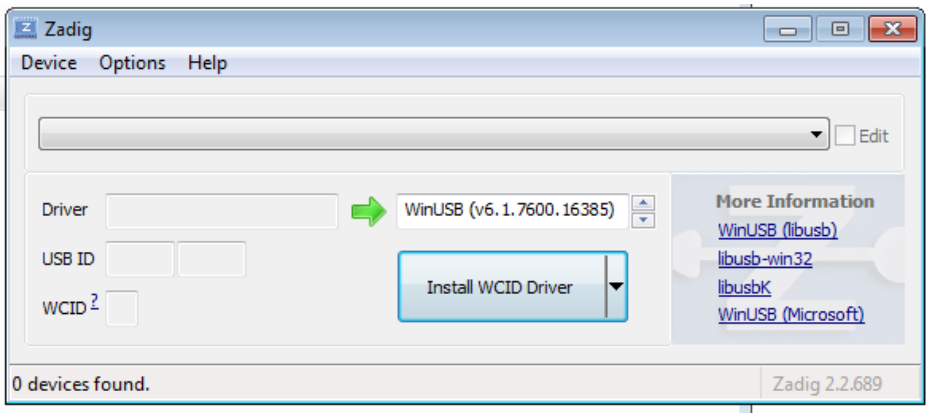
Upload the program to the PSoC and move onto the next step.

Software Requirements and Setup (Windows 7/10)

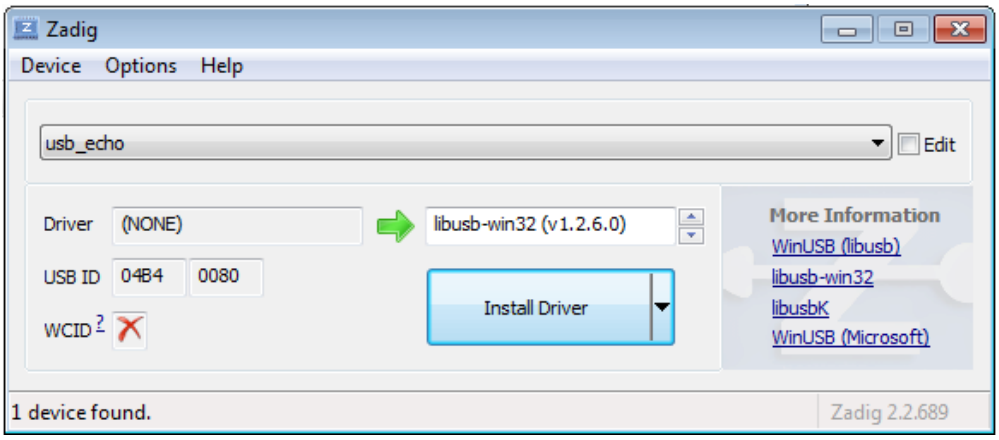
Ensure that you have Python 3 installed and available in your PATH (i.e. you can run python from the command line)

Besides Python, we are also going to need libusb (a C library for interfacing with USB) and PyUSB. We will use libusb to generate a generic driver for our device and then PyUSB and Python to communicate with it.

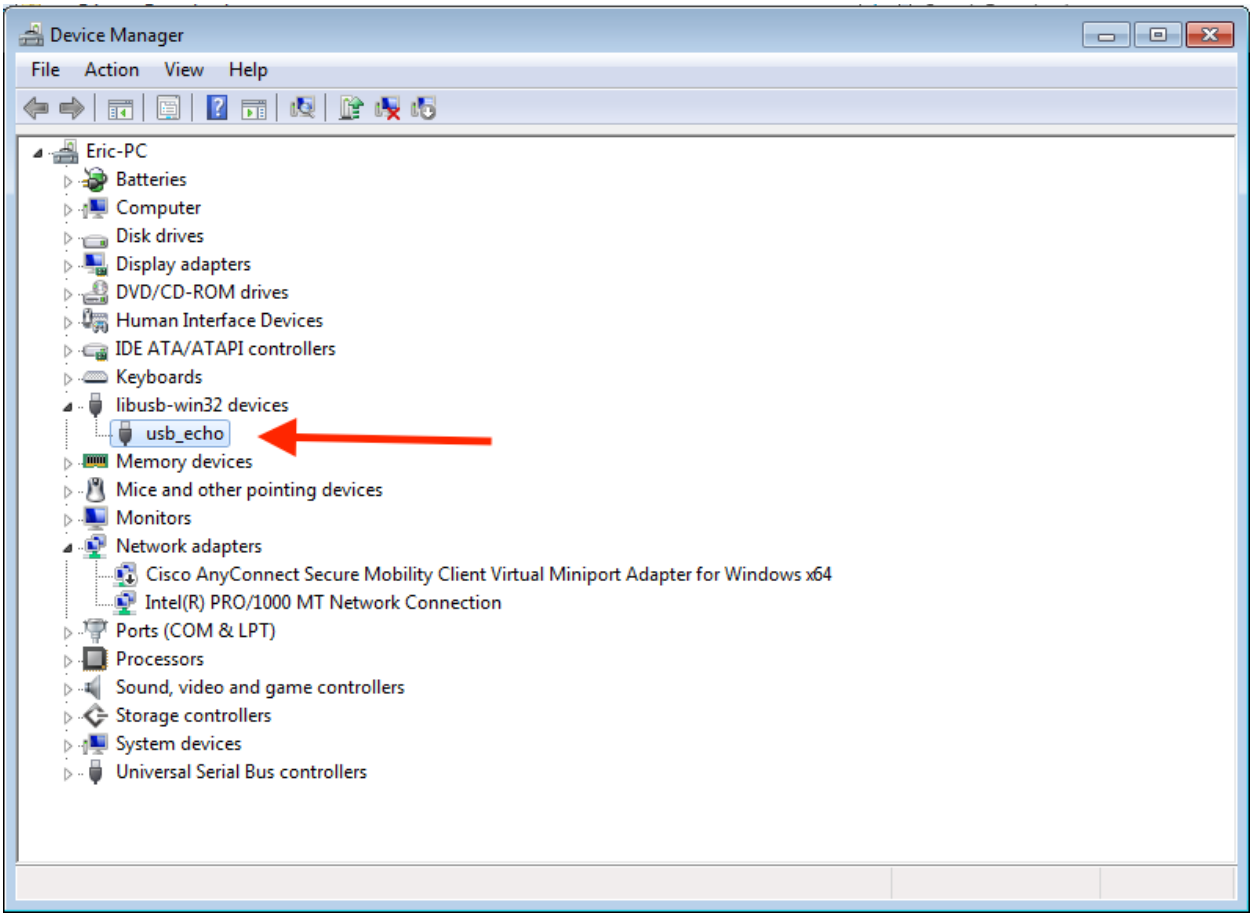
To generate the proper libusb driver, we will use a program called Zadig (http://zadig.akeo.ie/). Download it and run it to be presented the interface below:



Then, plug in your device though the USB port (generally not the same port used to program the device) and it should show up in Zadig. Set the driver type to libusb-win32 and press install.



Once the driver is installed, in the windows Device Manager (can be search for in the Start Menu), your device should show up under 'libusb-win32 device' if it installed correctly



Now that this device has been successfully installed, we need to install PyUSB. Fortunately, Python comes with a package manager called pip. To install pyusb simply enter the command `pip install pyusb` into your command line.

Software Requirements and Setup (Mac OS X)

For Mac OS X it is recommended to use a package manager such as homebrew (http://brew.sh). It will greatly simplify installation of the necessary requirements. This tutorial assumes you have installed homebrew

Install python and libusb using the command 'brew install python3 libusb' in your command line. Now that python is installed, you can use pip to install PyUSB. To install PyUSB, run the command 'pip install pyusb'.

Software Requirements and Setup (Linux)

This tutorial is assuming Ubuntu Linux, but Ubuntu's package manager can be substituted by your Linux distribution's package manager and associated command flags. Ubuntu uses 'apt-get' and the 'install' command line flag to signal installation of requested packages.

Install python3, pip, and libusb: 'sudo apt-get install python3 python3-pip libusb-dev'

Install pyusb using pip: 'sudo pip3 install pyusb'

Using Python to talk to the device

The following python code should run on all three platforms. The program activates the USB device assigned to our vendor and product ID and finds the IN and OUT endpoints. Then it waits for user input, sends the entered endpoint, and prints out the received input, assuming the received input is of the same size as the sent output. Certain Linux distributions and setups may require a 'sudo' before running the code to give python the necessary privileges to access USB devices.

Here is the python script:

```
import usb.core
import usb.util
import array

# search for our device by product and vendor ID
dev = usb.core.find(idVendor=0x4B4, idProduct=0x80)

#raise error if device is not found
if dev is None:
      raise ValueError('Device not found')

# set the active configuration (basically, start the device)
dev.set_configuration()

# get interface 0, alternate setting 0
intf = dev.get_active_configuration()[(0, 0)]

# find the first (and in this case only) OUT endpoint in our interface
epOut = usb.util.find_descriptor(
      intf,
      custom_match= \
      lambda e: \
            usb.util.endpoint_direction(e.bEndpointAddress) == \
            usb.util.ENDPOINT_OUT)

# find the first (and in this case only) IN endpoint in our interface
epIn = usb.util.find_descriptor(
      intf,
      custom_match= \
      lambda e: \
            usb.util.endpoint_direction(e.bEndpointAddress) == \
            usb.util.ENDPOINT_IN)

# make sure our endpoints were found
assert epOut is not None
assert epIn is not None

print("Message: ")
t = input() # get the user input
i = len(t)
epOut.write(t) # send it
print("Received: " + str(epIn.read(i))) # receive the echo
```

Remarks

- This tutorial borrows heavily from Craig Cheney's 2013 USB tutorial written for 6.115
- A more in-depth tutorial for using the PyUSB library (which uses the libusb backend) can be found at https://github.com/walac/pyusb/blob/master/docs/tutorial.rst
- There are various transfer types to explore that may be more suitable for your application. A good introduction to USB can be found at http://www.beyondlogic.org/usbnutshell/usb1.shtml
- DMA should also be explored as a more efficient way of moving large amounts of data into and out of the USB communication buffers.