CMPE 121L: Microprocessor System Design Lab

Spring 2019

Lab Exercise 5

Check-off Due in lab section, week of May 13, 2019

Report due: May 20

The purpose of this exercise is to learn how to transfer data over a USB 2.0 link set up between the PSoC5 and the Raspberry Pi boards. We will modify the same applications developed in Lab 4 to use the USB link instead of the UART. USB is better suited to transferring blocks of data at much higher speeds and has built-in protocols to support the communication needs of various applications.

The USB 2.0 Specification supports three speed classes: Low Speed (1.5 Mbits/second), Full Speed (12 Mbits/second), and High Speed (480 Mbits/second). We will use the Cypress USBFS component in the PSoC Creator library, which supports the Full Speed option. The PSoC board has a USB OTG (On-the-Go) port which can be configured as either USB host or USB device. In this exercise, we will use the USB port of the Raspberry Pi as the USB host and the OTG port of the PSoC as the USB device.

Each USB device is associated with a 16-bit Vendor ID and a 16-bit Product ID, which are used by software to identify the device. The Vendor ID we will use in this example is 0x04B4, which is assigned to Cypress Semiconductor, and the Product ID is 0x8051, which is not associated with a real product. The USB 2.0 Specifications define four distinct types of data transfers: **Control**, **Bulk**, **Interrupt**, and **Isochronous**. Each of these transfer types is designed to meet the needs of certain applications, and each is associated with its own protocols. In this exercise, we will use the **Bulk Transfer**, which is meant to be used for large block transfers (for example, for reading data from a disk). You may set up multiple transfers between the host and device and perform them concurrently (they will take turns on the physical link). Transfers of data from the host to device are called **OUT** transfers and those from the device to the host are called **IN** transfers. You may set up to 16 concurrent transfers in each direction. Each of these transfers needs to be distinguished with a 4-bit **Endpoint Address.**

Before starting this exercise, you should read the Cyprus application notes AN57294 and AN56377, which provide an excellent overview of USB protocols.

Using the USB Port on the Rapberry Pi: Setting up *libusb*

You can develop USB applications on the Raspberry Pi using the **libusb** library. This library has an extensive set of API functions, but you need to use only a few of them to design a basic application. The complete documentation on libusb can be found here.

The libusb library is pre-installed on your SD card, but you will need to update it with the following commands:

```
sudo apt-get install libusb-1.0-0
sudo apt-get install libusb-1.0-0-dev
```

You can include this library functions in your code by adding the following header line to your program.

```
#include <libusb.h>
```

You should then compile your code (for example, *myprog.c*) using the command line

```
gcc -o myprog myprog.c -I/usr/include/libusb-1.0/ -lusb-1.0
```

To verify that your software is installed correctly, download the test program *usb_discover.c* from the files/Lab Exercises area of the course Website into your Raspberry Pi, compile and run.

```
gcc -o usb_discover usb_discover.c -I/usr/include/libusb-1.0/ -lusb-1.0

sudo ./usb_discover
```

This code will scan the USB bus attached to the Raspberry Pi and print out the Vendor IDs and Product IDs of all the USB devices found (keyboard, mouse, etc.).

Bulk Transfer Example using *libusb*

Download the example code *usb_bulk_transfer.c* from the files/Lab Exercises area of the course Website into your Raspberry Pi and review the sequence of *libusb* calls in it. This program sets up two USB bulk transfers between the Raspberry Pi and a device (which will be implemented by the PSoC board, described later), an IN transfer and an OUT transfer, with Endpoint addresses 0x1 and 0x2, respectively.

The first step in the code is to initialize the libusb libray by calling *libusb_init()*. The application then opens the device with the known Vendor ID and Product ID using *libusb_open_device_with_vid_pid()*. This function returns a handle to the device data structure, which is used in the following function calls to identify the device. After opening the device, the application does a reset on the device with *libsub_reset_device()*. This step is optional, but helps to clear any residual state in the device (for example, caused by bugs in the code during testing). The application then enables the device using the *libusb_set_configuration()* function. It then needs to "claim" the interface for use by calling *libusb_claim_interface()*. The interface is now ready for use.

Data transfers are performed using the *libusb_bulk_transfer()* function. This function is used for both IN and OUT transfers, and the direction of the transfer is identified by the most significant bit of the second argument (0 for OUT transfers and 1 for IN transfers). The lower 4 bits of the second argument identify the Endpoint. Note that the actual number of bytes transferred may be

less than the requested size of the transfer, and multiple calls to the function may be needed to send the entire block of data. The code needs to check the value of the 5th argument when the function returns and make further calls if the value is less than the size the block remaining to be sent (this would normally occur only when the block size is larger than the maximum packet size, which is 64 bytes in our case). The code should also check the returned status for errors, and abort the program if a fatal error is detected.

The transfer implemented by *libusb_bulk_transfer()* is referred to as *synchronous IO*, because the function waits until an entire packet of data is delivered across the bus. In the case of an OUT transfer, it waits until an ack is received from the device before returning. This is simple to use, but is not very efficient. A more efficient option is to use *asynchronous IO*, which is based on API functions that do not block. These functions allow you to initiate multiple transfers and monitor their progress. You are not required to use asynchronous IO in this lab.

The final argument of *libusb_bulk_transfer()* is a timeout value for the function call. Setting it to 0 disables the timeout. Setting to a non-zero value prevents a hang of the program when the device is not ready, but

Using the OTG Port in PSoC

For this exercise, the USB device function will be implemented in the PSoC using the **USBFS** component. To get familiar with this component, open the code example *USBFS_Bulk_Wraparound*. This example implements two USB bulk transfers with the host, an IN transfer and an OUT transfer, and loops back all data received from the OUT transfer to the IN transfer (Note: do not modify this example to do the designs in Part 1 and Part 2. Instead, design them from scratch using this code as an example).

Open the data sheet for USBFS and review it, especially the API specification. The USBFS configuration shows two Endpoint Descriptors, the first defining an IN transfer (EndPoint 1) and the second defining an OUT transfer (EndPoint 2). The maximum packet size is 64 bytes, which is the maximum allowed for bulk transfer under the USB-2 Full Speed option. The Vendor ID and Product ID are part of the Device Descriptor.

Before your program can transfer data through the USB link, the USB component must be initialized and started using the API call

```
USBFS_Start(0u, USBFS_5V_OPERATION);
```

The `USFS_Start()` function calls `USBFS_Init()` to initialize the device and then starts its operation. The program must then wait for the host to enumerate the device and load its driver. This is achieved by the following code sequence:

```
while (!USBFS_GetConfiguration()){};
```

The program then enables the Endpoint for the OUT transfer (EndPoint address 0x2):

```
USBFS_EnableOutEP(0x2);
```

It now enters the main loop where it periodically polls the buffers in the USBFS component and loops back any data received from the host. In the main loop, it needs to continually check if the USB configuration has changed (as a result of unplugging the USB cable and plugging it back in), and re-enable the OUT Endpoint if the configuration has changed. This is achieved with the following code fragment. This fragment should always be included in the main loop.

```
if (USBFS_IsConfigurationChanged()){
   if (USBFS_GetConfiguration()){
      USBFS_EnableOutEP(0x2);
 }
}
```

To receive the data from the OUT transfer, the program first checks to see if the receiver buffer in the USBFS component is non-empty. It then queries the component for the number bytes received, and then reads the block of data into the user buffer.

```
if (USBFS_GetEPState(0x2) == USBFS_OUT_BUFFER_FULL)
 // check for non-empty buffer
 {
    length = USBFS_GetEPCount(0x2);
    USBFS_ReadOutEP(0x2, user_buffer, length);
 }
```

`USBFS_ReadOutEP()` is the function that performs the data move from the buffer inside the USBFS component to the user buffer. In the example, the transfer is performed through DMA, so `USBFS_ReadOutEP()` only sets up the DMA parameters and starts the DMA. The program needs to wait for the completion of the DMA before it can use the data in the user buffer. This is accomplished through the API call

```
while (USBFS_GetEPState(0x2) != USBFS_OUT_BUFFER_EMPTY){};
```

At this point, the data in the user buffer is ready for processing. The example code simply copies the data back into the USBFS component, but now into the transmit buffer used for the IN transfer. In general, the code should check if any previous data added to the buffer was delivered to the host, to avoid overwriting the buffer.

```
while (USBFS_GetEPState(0x1) != USBFS_IN_BUFFER_EMPTY){};
USBFS_LoadInEP(0x1, user_buffer, length); // Endpoint addr = 1
```

`USBFS_LoadInEP()` performs the data move from the user buffer to the buffer inside the USBFS component. Because the transfer is performed through DMA, `USBFS_LoadInEP()`

only sets up the DMA parameters and starts the DMA. The program needs to wait for the completion of the DMA before it can write more the data in the user buffer. This is accomplished by checking the buffer state before calling `USBFS_LoadInEP()`, as shown above.

Note that the USBFS component uses its own DMA controller to move data between its internal buffers and the main memory, and does not use the main DMA controller. Consequently, data can't be moved directly between the USBFS component and other peripherals. For example, to move data by DMA from an ADC to the USBFS component, it is necessary to set up two separate DMA transfers, one from the ADC to a memory buffer using the main DMA controller, and the second from the memory buffer to the USBFS component using the DMA controller associated with the USBFS component.

Testing the Loopback Application

Build the *USBFS_Bulk_Wraparound* project and program the PSoC board with it. Connect the PSoC board OTG port to the USB HUB attached to the Raspberry Pi. Run the *usb_discover*y program on the Raspberry Pi and verify that the device with Vendor ID 0x04B4 and Product ID 0x8051 is detected.

Compile and run the usb_bulk_transfer program on the Raspberry Pi.

```
gcc -o usb_bulk_transfer usb_bulk_transfer.c
-I/usr/include/libusb-1.0/ -lusb-1.0

sudo ./usb_bulk_transfer
```

The program will transmit 64 bytes to the device and display the data received back from the device.

Now you are ready to work on the exercise!

Part 1: Remote Control of LED brightness with PWM

This is the same design as in Part 1 of Lab 4, except that you will use a USB bulk transfer to transmit the data from the PSoC to the Raspberry Pi.
Create the design for the PSoC5 as follows:

1. Set up USBFS as in the loopback example. You will need to use only the IN transfer (EndPoint 1), because no data needs to be sent from the host to the device.
2. Add an 8-bit ADC to the design and wire up a potentiometer to apply a variable voltage at its input.
3. Write a PSoC5 program that samples the ADC every millisecond and transmits the byte through the USB link (In this case, treat the data as one-byte blocks).

Now design the Raspberry Pi side of your application.

1. First, connect one of the LED colors to pin 12 of the GPIO connector, which is one of the PWM-capable outputs of the Pi, through a current-limiting resistor.
2. Write a program to receive data from the USB link and drive the GPIO pin with the corresponding PWM setting. This can be done by modifying the example bulk transfer program.

Now connect the USB link and test your application. You may find it easier to start with a very low sampling rate (for example, once every second) to debug your code.

CHECKOFFS:

1. The potentiometer can change the LED brightness over its full range.
2. The Raspberry Pi code checks for errors on all USBFS -related system calls and aborts the program with a message when an error detected (similar to the code in the sample programs).
3. The application is able to run for many seconds with no errors.

Part 2. Analog Loopback through the Raspberry Pi

This is the same Part 2 of Lab 4, with the data transfer performed over USB. You will sample an analog signal continuously with the PSoC, convert the data to digital and send it over a USB bulk transfer to the Raspberry Pi. The Raspberry Pi will loop it back to the PSoC, which will then convert it to analog and output on an analog output pin.

First, design the PSoC side of the application as follows:

1. Set up USBFS as in the loopback example. You will need to use both the IN and OUT Endpoints for this part.
2. Add an 8-bit ADC to the design and connect its input to an unused GPIO pin.
3. Define two 64-byte character arrays in memory. Set up a DMA transfer from the ADC to the memory arrays. The descriptor chain should consist of two descriptors executed in a loop, each transferring 64 bytes to one of the arrays. The DMA should raise an interrupt at the end of each 64-byte block. Connect the DMA Request input to a clock and set its frequency such that the sampling rate of the ADC does not exceed the maximum sampling rate specified in the data sheet. Use the two character arrays as ping-pong buffers to transfer data to the USBFS component. After the first 64-byte buffer is filled by DMA, your program should send the bytes to the USBFS buffer for transmission on the link, while the DMA is filling the second buffer. When the second buffer is filled, the roles of the two buffers are switched, thus maintaining a continuous transfer of data. You should check for any data loss by making sure that the entire 64-byte buffer is emptied through the USB before the next buffer is completely filled. The program should turn on a LED if a data loss is detected (that is, if one of the ping-pong buffer becomes full while

the transfer of the other buffer through the USB link has not completed.). If there is data loss, you will need to decrease the sampling rate of the ADC.

4. Now design the receive side of your application. Add an 8-bit DAC to your design and connect its output to a GPIO pin.
5. Just as in the case of the ADC-to-memory transfer, define two 64-byte ping pong buffers. Set up a DMA channel with two descriptors to transfer data from the memory buffers to the DAC. In this case, the ping-pong buffers are filled from the USBFS component and are drained by the DMA to the DAC. To avoid data loss, you need to make sure that, on the average, data is drained by the DMA from the buffers at the same rate they are filled from the USB link. This can be done by driving the DMA Request input from the same clock used to drive the DMA Request input of the other channel.

Now design the Raspberry Pi side of your application. Write a program to receive data from the USB link continuously and transmit it back on the link. This can be done by modifying the code in the bulk transfer example. Now connect the USB link between the PSoC and Raspberry Pi and test your application.

Your design must be able to reproduce a 100 Hz sine wave at the DAC output with no significant degradation. Try to increase the frequency of you signal and test how far you can increase the frequency before the DAC output stops faithfully reproducing the signal. In your report, discuss how you can improve your design to support much higher frequencies, for example, 10 kHz.

CHECKOFFS:

1. The DAC output faithfully reproduces a 100 Hz sine wave applied on the ADC input without data loss and errors.
2. The Raspberry Pi code checks for errors on all libusb system calls and aborts the program with a message when an error detected (similar to the code in the sample programs).
3. The application is able to run for many seconds with no errors.

What to submit?

Your report must include:

- Schematics showing connections between the PSoC and Raspberry Pi.
- Descriptions of your designs
- Discussion of any problems encountered

Do not forget to submit the code for both the PSoC and Raspberry sides of the applications. Your code must be maintained in GitLab.