CMPE 121L: Microprocessor System Design Lab

Spring 2019

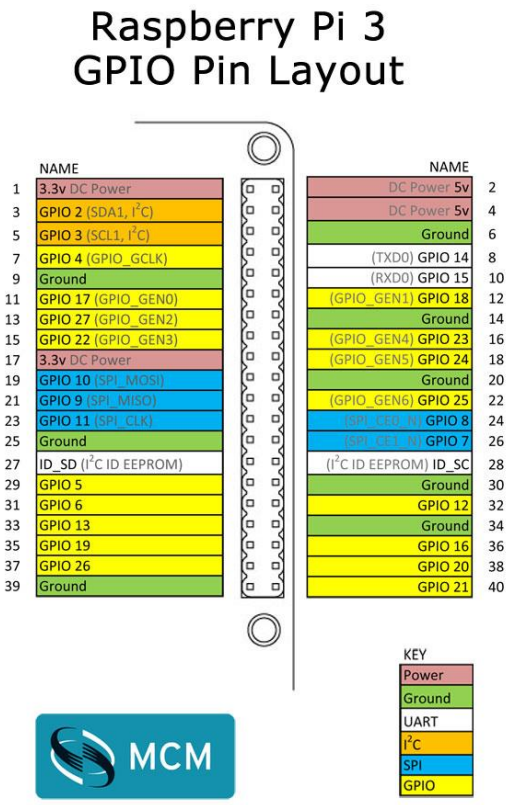
Lab Exercise 4

Check-off Due in lab section, week of May 6, 2019

Report due: May 13, 11:55 PM

The purpose of this exercise is to familiarize with the Raspberry Pi hardware and programming environment. You will learn how to use its UART as well as GPIO pins and perform basic communication and I/O. Before starting this exercise, read the document *Getting Started with Raspberry Pi* to help you configure the Raspberry Pi.

Raspberry Pi GPIO Interface



The layout of the Raspberry Pi GPIO connector is shown above. Some of the pins are dedicated to various hardware blocks (UART, SPI, I2C). All the remaining GPIO pins can be used for general-purpose IO and each can be configured as input or output. Note that the IO voltage is 3.3V, so the voltage applied on any of these pins should not exceed 3.3V.

Configuring the Raspberry Pi UART

To use the UARTs on the Raspberry Pi, you need to first enable serial interfaces. This can be done as follows:

1. Click on the Pi icon at the top left corner, select **Menu > Preferences > Raspberry Pi Configuration > Interfaces**, and enable serial interfaces.
2. Run raspi-config using the command

   sudo raspi-config

Navigate to **Interfacing Options > Serial** and select **No** to the question "Would you like a login shell to be accessible over serial?" and **Yes** to the question "Would you like the serial port hardware to be enabled?".

Reboot the system for the changes to take effect.

The Raspberry Pi 3 has two UARTs. There is a full-function hardware UART that is connected to the Bluetooth controller, and a Mini UART that is limited in functionality. Because we are not using the Bluetooth interface, it is possible to disconnect the former from the Bluetooth controller and connect its serial interface to the pins. Also, to obtain the correct baud rate, we need to set the clock frequency for the system clock. This is accomplished by adding the following lines to the file `/boot/config.txt` .

```
dtoverlay=pi3-disable-bt
core_freq=250
```

Note that you need superuser privileges to modify this file, so prefix your command with "sudo" when starting the editor. You will then need to reboot the system for the setting to take effect. The UART is now connected to pins 8 (TX) and 10 (RX) on the GPIO connector.

The Linux operating system treats the UART as a device and provides its own driver to manage it. In Linux, devices are treated similar to files, that is, the user first opens them, then performs reads/writes, and finally closes them. The operating parameters of the UART (its speed, data length, parity, etc.) are defined in a structure called **termios**, described in the Linux Programmer's Manual. You need to include the system library termios.h to be able to access this structure.

Your software can access the UART similar to a file named **/dev/serial0**. You need to first open it, just like a file and close it when you are done.

```
int fd = open("/dev/serial0", O_RDWR | O_NOCTTY | O_NDELAY);
```

This returns a descriptor to the file (-1 when the open fails). The second argument is a set of attributes that describes the desired options in using the device (see comments in example programs for details). The O_NDELAY option sets up the API as nonblocking, so that reads and writes to the device will never block.

At the end of the program, you need to close the device just as you would do to a file.

```
close(fd);
```

The **write** function allows you to transmit data on the interface:

```
wrcount = write(fd, txbuffer, length);
```

txbuffer is a character array containing the bytes to be transmitted and **length** is the number of bytes to be transmitted. The return value is the actual number of bytes transferred to the UART and can be smaller than **length** (this occurs when the UART transmit buffer has insufficient room to store all the data at the time when the function was called). If the return value is smaller than length, you will need to call the write function again to transmit the remaining data. For example, assume **txbuffer** is an array of size 100 and the function is called with the length parameter set to 100. If the return value is 80, then only the first 80 elements of the array have been transmitted, and you need to call the function again with length set to 20 to transmit the elements **txbuffer[80]** through **txbuffer[99].**

If the return value of the write function is negative, it indicates failure, but it is important to distinguish two scenarios where this can occur:

1. The transmit buffer of the UART (within the kernel) is currently full and the function was unable to transmit any data. This is not a fatal error. In this case we just need to retry the write, which will eventually return successfully when there is room in the buffer.
2. There was a fatal error, which requires aborting the program.

These two cases are distinguished by checking the **errno** system variable for the return error code. A return error code of **EAGAIN** indicates the transmit buffer full condition (see how the return status check is handled in the example **pi_uart_tx.c**).

Your program can read the received data from the UART using the **read** function.

```
rdcount = read(fd, rxbuffer, sizeof(rxbuffer));
```

rxbuffer is a character array where you want the data to be transferred to. The return value is the actual number of bytes transferred into the array. Because the function is nonblocking, the return value can range from 0 to the maximum size (specified by the third argument). Thus, your program normally needs to keep calling this function until all data is received. A negative return value indicates a fatal error.

Before writing your code for Part 1, familiarize yourself with the two example programs **pi_uart_tx.c** and **pi_uart_lpbk.c**. The first program transmits a fixed string repeatedly. The second program transmits the string "Hello World", receives it back, and prints it. Run the first program, observe the TX data on pin 8 on an oscilloscope and verify that the UART settings (115200 baud, 8-bit data, odd parity) are reflected in the waveform (confirm that the baud rate is correct). Then connect pin 8 to pin 10 using an external jumper wire and run the **pi_uart_lpbk** test to verify that the RX side is working correctly.

Connecting the PSoC5 and Raspberry Pi GPIO pins

When connecting a PSoC5 GPIO output to a Raspberry Pi GPIO input, you need to make sure that that voltage on the Pi GPIO input does not exceed 3.3V. This can be done in two ways: (i) Use a voltage divider to reduce the 5V output to 3.3V, or (ii) power the VDDIO voltage rail of the PSoC5 board with 3.3V power from the Raspberry Pi GPIO connector. The second option requires separating the VDDIO rail from VDD by removing a zero-ohm resistor on the board (see handout on getting started with the PSoC for details).

Part 1: Remote Control of LED brightness with PWM

In this part, you will control the brightness of a LED connected to one of the GPIO outputs of the Raspberry Pi using PWM. The BCM2837 microcontroller on the Raspberry Pi board does not have built-in analog blocks, so we can't connect a potentiometer to it directly and use it to control the LED brightness. Instead, we will connect the potentiometer to the PSoC board and send its setting to the Raspberry Pi through a UART link.
First, create a design for the PSoC5 as follows:

1. Add a UART to the design with the following parameters: 115200 baud, 1 stop bit, odd parity, no hardware flow control. You may select the interrupt options, buffer size, etc., based on your design preferences.
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 UART.

Verify your program by observing the output of the UART on an oscilloscope.

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. Initialize WiringPi and set up the GPIO pin as a PWM output.

   ```
   #define LED_PIN    1
   wiringPiSetup ();
   pinMode (LED_PIN, PWM_OUTPUT);
   ```

   Note that the pin identifiers to be used in WiringPi functions are not the same as the GPIO pin numbers. See *Gettind Started with Raspberry Pi* for a translation of the pin numbers on the GPIO connector to the WiringPi pin identifiers.

3. Write a program to receive data from the UART continuously and drive the GPIO pin with the corresponding PWM setting. You can control the pin using the WiringPi function

   ```
   void  pwmWrite (int pin, int value);
   ```

where the pin identifier is LED_PIN and **value** controls the duty cycle of the PWM signal. Because its range is 0–1023, you need to scale the incoming byte by multiplying by 4 to use in this function.

Now connect the transmit data from the PSoC UART to the receive side of the Raspberry Pi UART and test your application. Do not forget to connect the grounds of the two systems together if they are not powered from the same power supply.

CHECKOFFS:

1. The potentiometer can change the LED brightness over its full range.
2. The Raspberry Pi code checks for errors on all UART-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

In this part, you will sample an analog signal with the PSoC, convert the data to digital and send it over a UART link to the Raspberry Pi. The Raspberry Pi will loop it back to the PSoC, which will the convert it to analog and output on an analog output pin. Thus, you will have created the equivalent of a long wire between an input pin and an output pin of the PSoC.

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

1. Set up the PSoC5 UART with the following parameters: 115200 baud, 1 stop bit, odd parity, no hardware flow control. You may select the interrupt options, buffer size, etc., based on your design preferences. Do not set the UART buffer size larger than 4.
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 UART. After the first 64-byte buffer is filled by DMA, your program should send the bytes to the UART 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 convince yourself that this scheme does not cause any data loss by making sure that the entire 64-byte buffer can be emptied through the UART before the next buffer is completely filled. This depends on the sampling rate of the ADC and the baud rate setting of the UART. If there is a possibility of data loss, you will need to either decrease the sampling rate of the ADC or increase the baud rate of the UART to fix the problem. It is recommended that you introduce a check in your code to ensure that each 64-byte buffer

has been emptied through the UART before the DMA starts to transfer data into it, and signal any data losses by turning on an error LED.

4. To test your program, connect the UART TX output to the AD2 logic analyzer and capture the serial data stream. If the voltage on the ADC input is held steady, you should be able to see a pattern of serial data with the same byte value.
5. Now design the receive side of your application. Temporarily loop the TX output of the UART to its RX input and set the RX side to generate an interrupt when its FIFO is non-empty.
6. Add an 8-bit DAC to your design and connect its output to a GPIO pin.
7. 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 RX FIFO of the UART 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 by the UART. 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.
8. Test your program by looping back data from the TX pin of the UART to its RX pin. Connect the ADC input to the AD2 pattern generator and the DAC output to an oscilloscope. Set the AD2 pattern generator to generate a 100 Hz sine wave. The DAC output should faithfully reproduce the waveform.
9. Increase the frequency of the waveform and check how far you can go before the design fails to reproduce the waveform on the output pin. Note the maximum frequency in your report and discuss what limits the speed and how the design can be improved.

Now design the Raspberry Pi side of your application. Write a program to receive data from the UART continuously and transmit it back on the link. Now connect the UART link between the PSoC and Raspberry Pi and test your application. Do not forget to connect the grounds of the two systems together if they are not powered from the same power supply.

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 UART-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.

What to submit?

Your report must include:

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

Do not forget to submit the code for both the PSoC and Raspberry sides of the applications.