Mechtron/Sfwr Eng 4AA4 - Lab 8

Resource sharing protocols and controlling servo motors

Introduction:

Pulse Width Modulation (PWM) is a term used to describe a type of digital signal that can result in a varying output voltage. Although a digital signal can only be high (usually 5V or 3.3V) or low (ground) at any time, one can change the *proportion* of time the signal is high compared to the time when it is low over a consistent time interval. The duration for which the signal is high is called "On Time" and is usually described as a percentage of the time period of the signal for which it is high. This percentage is called " Duty Cycle". For example a digital signal that is high for half of its time period has a duty cycle of 50%. Average value of the signal varies according to its duty cycle.

The position of a servo motor can be controlled by using a PWM control signal where there is a 0.2 to 2 ms high impulse every 20ms. The impulses′ duration corresponds with the position of the motor where:

- 0.2ms is full left (0 degree)
- 1.1ms is the middle (90 degrees)
- 2ms is full right (180 degrees)

Goals:

- Design and implement a real time task to generate a PWM signal on one of the myRIO pins.
- Use the PWM signal to control the position of a servo motor.
- Learn about and implement FIFO buffers within Linux for inter-process communication.
- Learn about and implement shared memory for inter-process communication.

Reference materials:

Please read the following reference materials before lab session starts. Some of the following materials can be found in the `ref` folder after you download and unzip the `lab8.zip` file.

- MyRIO_Shipping_Personality_Reference6.0.pdf.
  The section about PWM. Page 22 and 23 are going to be very useful.
- MyRio.pdf
  For myRIO-1900, only some specific pins have PWM functions. Please refer to page 4 and page 6 to figure out which pins you will need to use for this lab.
- https://www.softprayog.in/programming/interprocess-communication-using-fifos-in-linux

- http://www.tldp.org/LDP/lpg/node18.html
  These two references are for Part 2 of this lab, which will use the FIFO buffer.

- shared_memory.pdf (from: http://home.deib.polimi.it/fornacia/lib/exe/fetch.php?media=teaching:piatt_sw_rete_polimi:unix-shm.pdf)
  Part 3 of this lab will explore the shared memory. These slides can give you a quick tutorial for how to implement and use shared memory on Unix/Linux system.

Part 1: Servo motor position control [35]

The frequency (or the time period) of the PWM signal generated by myRIO-1900 is determined by the value set in the `PWM Maximum Count Register` and the **clock divider** which is determined by the value set in the `PWM Clock Select Register`. With the **MODE** bit in the `PWM Configuration Register` being set to 1, the PWM counter would repeat cycles of incrementing from 0 to the value specified in the `PWM Maximum Count Register` then resetting to 0, at a certain clock rate. The frequency of these cycles is the PWM frequency.

By default, the PWM counter and other myRIO-1900 hardware run at the base clock frequency, which is 40MHz. With a clock divider N, which is determined by the value set in the `PWM Clock Select Register`, The PWM counter will run at a lower clock rate of (40/N) MHz.

The following formula can be used to calculate your PWM frequency:

$$f_{PWM} = \frac{f_{clk}}{N(X+1)} \tag{1}$$

where:

f_{PWM} is the frequency of the PWM signal generated by myRIO-1900.
N is the **clock divider** whose value is determined by the settings in the `PWM Clock Select Register`.
X is the value set in the `PWM Maximum Count Register`.

The duty cycle of the PWM signal is determined by the **compare value** set in the `PWM Compare Register`. For example, with proper settings in the `PWM Configuration Register`, when PWM counter counts from 0 to the **compare value**, the output on the PWM pin can be set to 1, once the PWM counter equals the **compare value**, the output on the PWM pin can be cleared, hence gives a certain period of "on time" and a certain duty cycle.

1. Create a new project using the myRIO template. The sample project "`myRIO Example--PWM`" can be used as a guide for the coming steps.

2. Enable PWM on the board using the function "`PWM_Configure( )`".

   - The configuration should be done by using a "`MyRio_PWM`" structure, which is defined in "`PWM.h`".

- Make sure take notes of your configuration–you should configure the **PWM0** on MXP port A.

  Note: if you get errors saying PWMA_xxxx is NOT found, you can simply ignore them, or you can delete the errors, clean the project, and rebuild it. Also make sure that "`PWM.h`" is included.

3. The output pins of PWM are shared with other on board devices. It is therefore necessary to select the PWM on the appropriate SELECT register (use function `NiFpga_WriteU8()`).

4. For the servo motors used in our lab, the PWM time period must be 20ms, which corresponds to a frequency of 50Hz. According to Equation 1, this can be achieved by choosing different combinations of **clock divider**s and the **maximum count** values. We can use the clock divider "`Pwm_16x`" in our lab.

   - With the **clock divider** to be 16, the PWM counter increments at 40MHz / 16 = 2.5MHz. If set the **maximum value** of the PWM counter to 49,999, the frequency of the PWM signal, i.e., the frequency that the counter counts from 0 to 49,999 will be 2.5MHz / 50,000 = 50Hz.
   - To position the servo motor to its left most position, a 0.2ms high impulse is needed. With the selected clock divider, counting 500 ticks will take the PWM counter 0.2ms. This means setting the **compare value** to 499 in the `PWM Compare Register` will generate a PWM signal to position the servo motor to its full left position.
   - Setting the **compare value** to 4999 in the `PWM Compare Register` will generate a PWM signal to position the servo motor to its full right position.

5. Code your program to prompt users to input the desired motor position in degrees between 0 to 180. Coerce any input values that are out of the range to either 0 or 180 degree.

6. Attach the servo motor to the appropriate myRIO pins. The servo motor wires are: Red: +5V, Black: GND, Yellow/White: PWM Signal.

7. Build and run your project on myRIO. Test your program to make sure you can position the servo motor to 0, 90 and 180 degrees. Demonstrate your working project to one of the TAs. **Note: Save a copy of this part of work for your bonus part later**.

Part 2: Use FIFO to pass values to control motor position [50]

- Part 1 focused on just changing the position of the motor. In part 1, the motor control application takes inputs directly from users and use the inputs to control the servo motor.

- Linux provides a mechanism called FIFO, which is a unidirectional buffer that can be used for inter-process communication. In this part, we will implement a user application to take input from the user, and then use a FIFO to pass the input to a motor control application.

- You should use external sources (Google) to learn more about the Linux FIFO buffer. The two references in the **Reference materials** section are great resources.

- A very barebone example program, `FIFOSample.c`, for creating and using FIFO can be found in the `data` folder in `lab8.zip` file. The `FIFOSample.c` is not complete, but can be used as a starting point.

- **Please NOTE 1:** By default, blocking will occur on a FIFO. In other words, if a FIFO is opened for reading, the process will be blocked until some other processes open the FIFO for writing. if a FIFO is opened for writing, the process will be blocked until some other processes open the FIFO for reading. When you code and test your user and motor control applications in this lab part, sometimes you may see your programs being "blocked".
- **Please NOTE 2:** If you do not want your application to be blocked by the above mentioned FIFO nature, the **O_NONBLOCK** flag can be used in an `open()` call to disable the default blocking action (not sure for `fopen()` call). Using this flag may help you when you debug your client (user application) and server (motor control) application separately. **HOWEVER**, it will be tricky to do this lab if you use this flag. Therefore, you are recommended **NOT TO USE the *O_NONBLOCK* flag** in this lab although it may help you when debug part of your program, unless you have interest to explore around.

1. Create a new project from the myRIO Template. Use code modified from Part 1 so that a FIFO is created and the desired motor position is read from the FIFO buffer. The value in the FIFO will be written by a user application program.
2. Create a new project from the template of Hello World. This application will open the FIFO created by the motor control application, will read a desired motor position from the user and write this user input into the FIFO.
3. Build and run these two projects. By running the projects, the build binaries are downloaded to the myRIO to a specific folder with specific names, which are specified by the Eclipse's **Run Configuration**.
4. Use a SSH client such as ***Bitvise SSH Client*** to connect to the myRIO. Open two terminals.
5. In one terminal, navigate to the appropriate folder, launch your motor control application (do not forget to prefix a "./" before the binary name). The application now should be waiting to read a value from a FIFO.
6. In another terminal, launch your user application which allows you to input a value for desired motor position.
7. Test and debug your applications. Demonstrate your working projects to one of your TAs.

Part 3: Use shared memory to pass values to control motor position [15]

- Another inter-process communication method is to use shared memory. In this part of the lab you are required to modify your code to store and read data from shared memory instead of passing it through the FIFO buffer.
- The example code in the shared_memory.pdf slides uses a **struct** type variable. Your project may just need an integer or a float type variable.

- To use the shared memory, both the client and the server applications need to use `shmget(), shmat()` and other related functions, but the variable names to hold the returned values from these functions may be different. The **key** for the shared memory NEEDs TO BE THE SAME in the applications using this shared memory.

1. Follow the steps in Part 2 to create two projects, or copy the two projects from Part 2.
2. Modify your motor control application to read the desired motor position from the shared memory.
3. Modify your user application project to make the user input to be stored in shared memory.
4. Build and test your applications. Demonstrate your working projects to one of your TAs.

Bonus [discretionary]

Modify your code from part 1 to make the servo motor moves from the minimum position (0 degrees) to the maximum position (180 degrees) and back to the minimum position (0 degrees) continuously. This is known as a “sweep” function.