ME 477 Embedded Computing
Laboratory Experiment #4
Parallel Input/Output and Control

Objectives

The objectives of this exercise are to:

1. Become familiar with optical encoding.
2. Implement a finite state machine control algorithm.
3. Understand pulse-modulation control of a dc motor.
4. Use instruction timing to produce a calibrated delay.

Introduction

In this exercise, your program will control and monitor the speed of a dc motor. The interface between the myRIO and the motor will be digital input/output (DIO) channels, and an FPGA encoder counter. See Figure 1.

Pulse-Width Modulation—Channel-0 of connector-A ($\overline{\text{run}}$) is connected to a current source amplifier such that when $\overline{\text{run}}$ = 1, no current flows through the motor; and when $\overline{\text{run}}$ is 0, 0.25 amps flows. Your program will periodically alter this bit, applying an oscillating power signal to the motor. The *duty cycle* (the percentage of time power is applied) equals the percentage of time the channel is low.

Encoder/Counter—An optical encoder is mounted on the shaft of the dc motor. The encoder sends 2000 state changes per revolution. Therefore, each encoder state change corresponds to a motor rotation of 1/2000 revolution, called a Basic Displacement Increment (BDI).

An encoder counter in FPGA interface determines the total number of these state changes. The speed is determined by computing the number of state changes from the encoder during a certain time interval, called the Basic Time Interval (BTI). Therefore, the number of state changes occurring *during each interval* represents the angular speed of rotation in units of BDI/BTI.

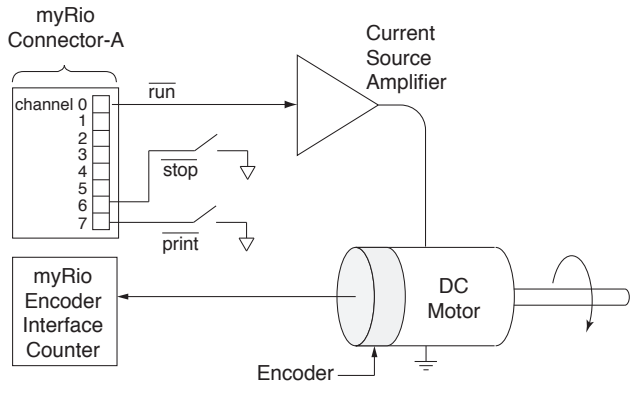


Figure 1: Pulse-modulation & speed measurement

Initializing the Encoder Counter—Counting of the encoder state changes is accomplished by the FPGA (Field Programmable Gate Array) associated with the Xilinx Zynq-7010 System-on-a-Chip, with dual Cortex A-9 ARM processors. The counter must be initialized before it can be used. Initialization includes identifying the encoder connection, setting the count value to zero, configuring the counter for a quadrature encoder, and clearing any error conditions. A function, included in the ME477Library, alters the appropriate control registers to initialize of the encoder interface on connector-C.

The prototype for the initialization function is:

```
NiFpga_Status
EncoderC_initialize( NiFpga_Session myrio_session,
                     MyRio_Encoder *channel);
```

The first argument, `myrio_session`, (type: `NiFpga_Session`) identifies the FPGA session, and must be declared as a *global variable* for this application. That is,

```
NiFpga_Session myrio_session;
```

The second argument `channel` (type: `MyRio_Encoder *`) points to a structure that maintains the current status and count value, and must also be declared as a global variable. We will use encoder #0. For example,

```
MyRio_Encoder encC0;
```

Reading the Encoder Counter—The position of the encoder (in BDI) may be found at any time by reading the counter value. The prototype of a library function provided for that purpose is:

```
uint32_t Encoder_Counter(MyRio_Encoder* channel);
```

where the argument is the counter channel declared during the initialization, and the returned value is the current count in the form of a 32-bit integer.

Pre-Laboratory Preparation

Main Program—Write a main program that produces a periodic waveform for $\overline{\text{run}}$ that applies an average current to the motor determined by the duty cycle. The period (1 BTI) will be N "wait" intervals, of which current will pass through the motor during M wait intervals ($0 < M < N$). See the first graph in Figure 2.

In addition, while channel-7 of connector-A is 0, the program will print the measured speed on the display at the beginning of each BTI. You will control channel-7 through a push button switch. The corresponding $\overline{\text{run}}$ waveform is shown in the second graph of Figure 2.

The algorithm should be implemented as a finite state machine. (See the notes on Finite State Machines.) As shown in Figure 3, the machine will have four possible states: "high", "low", "speed", and "stop". The inputs will be the **Clock** variable, and channels 6 and 7. The outputs will be $\overline{\text{run}}$, **Clock** (which may be reset to 0), and printing the **motor speed** on the LCD display.

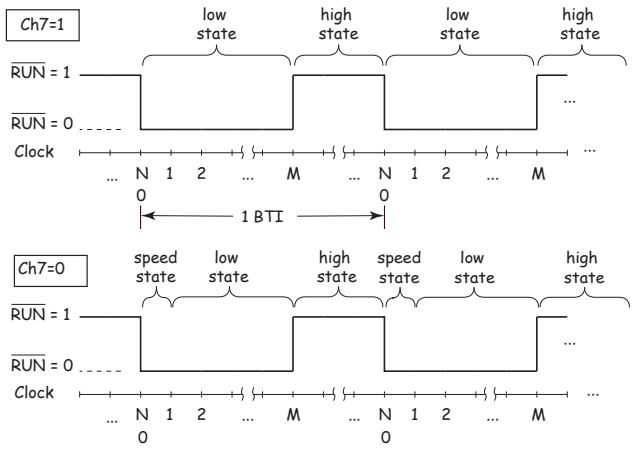


Figure 2: $\overline{\text{run}}$ waveforms.

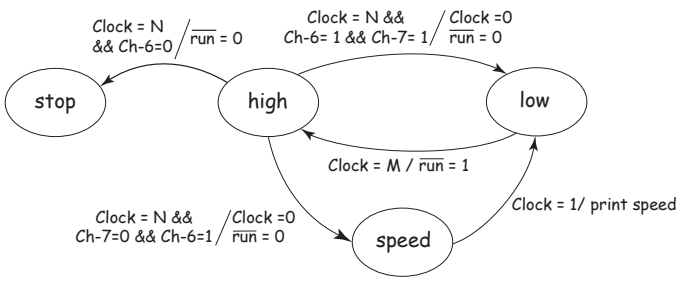


Figure 3: State transition diagram.

And, the corresponding state transition table, listing all possible transitions, is:

Current State	Inputs			Outputs			Next State
	$\overline{\text{stop}}$ Ch6	$\overline{\text{print}}$ Ch7	clk	$\overline{\text{run}}$ Ch0	clk	speed	
high	1	1	N	0	0	nc	low
high	1	0	N	0	0	nc	speed
high	0	x	N	0	0	nc	stop
low	x	x	M	1	nc	nc	high
speed	x	x	1	nc	nc	print	low
stop	x	x	nc	nc	nc	nc	exit

nc = no change in output. x = Dont care input.

Overall the main program will:

1. Use `MyRio_Open()` to open the myRIO session, as usual.
2. Setup all interface conditions, and initialize the finite state machine. See below `initializeSM()`.
3. Request, from the user, the number (N) of wait intervals in each BTI.
4. Request the number (M) of intervals that the motor signal is "on" in each BTI.
5. Start the main state transition loop.
6. When the main state transition loop detects that the current state is `exit`, it should close the myRIO session, as usual.

Functions

double_in() To execute the user I/O you may use the routine (`double_in`) developed in Lab #1, or you may simply call the function (from the me477library):

```
double  double_in(char *string);
```

initializeSM() Perform the following:

1. Initialize channels 0, 6, and 7 on connector-A according to Figure 1. For example, for channel-7,
```
Ch7.dir = DIOA_70DIR;
Ch7.out = DIOA_70OUT;
Ch7.in = DIOA_70IN;
Ch7.bit = 7;
```
2. Initialize the encoder interface. See above.
3. Stop the motor. $\overline{\text{run}} = 1$
4. Set the initial state to low.
5. Set the **Clock** = 0.

high() If the **Clock** is N { set **Clock** = 0 and $\overline{\text{run}} = 0$.
If Ch-7 = 0, change the state to **speed**.
If Ch-6 = 0, change the state to **stop**.
Otherwise, change the state to **low**.
}

low() If the **Clock** is M, set $\overline{\text{run}} = 1$, and change the state to **high**.

speed() Call **vel()**. The function **vel()** reads the encoder counter and computes the speed in units BDI/BTI. See **vel()** below. Convert the speed to units of revolutions/min. Print the speed as follows: "`printf_lcd("\fspeed %g rpm",rpm);`"

Finally, change the state to **low**.

double vel(void); Write a function to measure the velocity. Each time this subroutine is called, it should perform the following functions. Suppose that this is the start of the nth BTI.

1. Read the current encoder count: c_n (interpreted as an 32-bit signed binary number; `int`).
2. Compute the speed as the difference between the current and previous counts: $(c_n - c_{n-1})$.
3. Replace the previous count with the current count for use in the next BTI.
4. Return the speed `(double)` to the calling function.

Note: The first time `vel()` is called it should set the value of the previous count to the current count.

stop() The final state of the program.

1. Stop the motor. That is, $\overline{\text{run}} = 1$.
2. Clear the LCD and print the message: "stopping".
3. Set the current state to `exit`. The `while()` loop in `main()` should terminate if the current state is `exit`.
4. Save the response to a MATLAB file. See below Laboratory Procedure.

wait() Your program will determine the time by executing a calibrated delay-interval function. Consider this "wait" function.

```
/*-----------------------------------
Function wait
      Purpose:      waits for xxx  ms.
      Parameters:   none
      Returns:      none
*-----------------------------------*/
    void    wait(void) {
            uint32_t i;

            i = 417000;
            while(i>0){
                    i--;
            }
            return;
    }
```

Notice that the above program does nothing but waste time! The compiler generates the following operation codes for this function. The first column contains the addresses, and the second contains the corresponding opcodes.

```
wait+0        push {r11}
wait+4        add r11, sp, #0
wait+8        sub sp, sp, #12

wait+12        mov r3, #417000
wait+16        str r3, [r11, #-8]
wait+20        b 0x8ed4 <wait+36>

wait+24        ldr r3, [r11, #-8]
wait+28        sub r3, r3, #1
wait+32        str r3, [r11, #-8]
wait+36        ldr r3, [r11, #-8]
wait+40        cmp r3, #0
wait+44        bne 0x8ec8 <wait+24>

wait+48        nop ; (mov r0, r0)
wait+52        add sp, r11, #0
wait+56        ldmfd sp!, {r11}
wait+60        bx lr
```

The clock frequency of our microcomputer is 667 MHz.[1] Note carefully how the branch instructions are used. Determine the exact number of clock cycles[2] for the code to execute, accounting for all instructions. From that, calculate the delay interval in milliseconds.

When free running, the speed of the motor is approximately 1500 RPM. Considering all the above (note that the counter limit is 12 bits), determine a reasonable value for N, the number of delay intervals in a BTI. What inaccuracies or programming difficulties are there in using a delay routine for control and time measurement?

header files The following header files will be required by your code.

```
#include <stdio.h>
#include "Encoder.h"
#include "MyRio.h"
#include "DIO.h"
#include "me477.h"
#include <unistd.h>
#include "matlabfiles.h"
```

Modulo Arithmetic

We will estimate the rotational speed by computing the difference between the current encoder count c_n and the previous count c_{n-1}. The counter is capable of counting up and down, depending on the direction of rotation. Interpreting the count as 32-bit signed binary, the value is in the range $[-2^{31}, +(2^{31}-1)]$. For example, starting from 0 and rotating in the clockwise direction, the count will increase until it reaches $+(2^{31}-1)$, then "roll over" to -2^{31}, and continue increasing.

How will this rollover affect our estimate of the velocity? Assume that the current and previous counts (c_n and c_{n-1}) are assigned to signed integer variables of width equal to that of the counter. For our C compiler the `(int)` data type is 32 bits (8-bytes). Further assume that the angular position of the encoder changes less than $2^{32}/2000$ revolutions (about 2 million revolutions!) during a single BTI. That is, $|c_n - c_{n-1}| < 2^{32}$.

When we compute the difference between two signed integer data types, the result is defined by the *offset modulo* function:

$$\text{mod}\,(m, n, d) = m - n\left\lfloor \frac{m-d}{n} \right\rfloor \qquad (1)$$

where m is the value, n is the modulus, d is the offset, and $\lfloor x \rfloor$ is the "floor" function (i.e. the greatest integer less than or equal to x.) The result is modulo-n, and always in the range $[d, d+n-1]$.

Then, for our case of `(int)` data, we estimate the relative displacement using modulo 2^{32}, with offset $d = -2^{31}$.

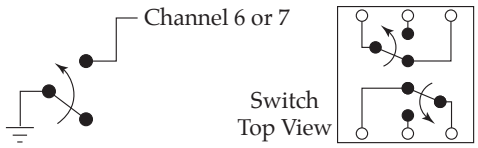
$$\begin{aligned} \Delta\theta &= \text{mod}\,(c_n - c_{n-1}, 2^{32}, -2^{31}) \\ &= c_n - c_{n-1} - 2^{32}\left\lfloor \frac{c_n - c_{n-1} - (-2^{31})}{2^{32}} \right\rfloor \end{aligned} \qquad (2)$$

Let's examine what happens when we cross the rollover point. Suppose that the previous counter value c_{n-1} was $2^{31}-2$. And, that during the BTI the encoder has moved forward by +4, such that the current reading c_n is $-2^{31}+2$. The numerical difference $c_n - c_{n-1}$ is -4,294,967,292. However, applying equation 2, the 32-bit signed integer arithmetic gives the correct result: $\text{mod}\,(-4,294,967,292, 2^{32}, -2^{31}) = +4$.

[1] Ref. myRIO User Guide and Specification
[2] Ref. Cortex-A9 Technical Reference Manual

Laboratory Procedure

1. Examine the circuit on the breadboard on connector-A of the myRIO. The push button switches shown below, connect channels 6 and 7 to ground when pressed. *Note:* These channels have pull-up resistors.



2. Use the oscilloscope to view the waveform produced by your program. For example, use N = 5, M = 3.
3. Compare your program's printed output (on the LCD display) to the steady-state motor speed as measured with the optical tachometer.
4. Use the oscilloscope to view the start/stop waveform produced by your program, and to measure the actual length of a BTI. Is it what you expect? If not, why not?
5. Repeat the previous step while printing the speed (press the switch). What does the oscilloscope show has happened to the length of the BTI. What's going on!?
6. Describe how you made this measurement and discuss any limitations in accuracy. In a later lab we will find ways of overcoming this limitation.
7. Recording a Step Response—After you have your code running as described above, try this: Record the velocity step response of the DC motor, save it to a file and plot it in MATLAB. Here's how:

   Add code to your **speed()** function to save the measured speed at successive locations in a global buffer. You will need to keep track of a buffer pointer in a separate memory location. Increment the buffer pointer each time a value is put in the buffer. The program must stop putting values in the buffer when it is full. For example,

```
#define IMAX    2400          // max points
static  double  buffer[IMAX]; // speed buffer
static  double  *bp = buffer; // buffer pointer
```

   and, in the executable code,

```
if (bp < buffer+IMAX) *bp++ = rpm;
```

   To record an accurate velocity, temporarily comment-out the `printf_lcd()` statement in **speed()**, and hold down `Ch7` switch while you start the program.
8. Saving the Response—The program should save the response stored in the buffer to a MATLAB (.mat) file on the myRIO under the real-time Linux operating system during the `stop` state. See the notes: "*Saving myRIO C data to a MATLAB file*" on the course website.

   The MATLAB file must be called `Lab.mat`. In the file save the speed buffer, the values of `N` and `M`, and a character string containing your name. The name string will allow you to verify that the file was filled by your program.

   The array can be plotted using the plot command.

1. From your plot, estimate the time constant of the system. Plotting points, instead of a continuous line, will make interpretation easier
2. What is the steady state velocity in RPM?

Extra

Fixing the M = 1 Case—You may have noticed that when M = 1 the finite state machine does not function as desired. What is wrong? How would by modify the state transition diagram to correct this problem? How would you modify the state transition table? Modify your program to correct the M = 1 case. Test the result.