04.L Lab Exercise: Finite state machine motor control

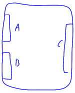
Objectives

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 drive and monitor the speed of a dc motor using a finite state machine model. The myRIO will drive the motor with pulse-width modulation (PWM) on a DIO channel configured as a digital output. This digital signal will be amplified by the analog amplifier described in Lec. 04.2 , as shown in Fig. L.1. The speed of the motor will be measured with a quadrature encoder on the motor and read by the myRIO FPGA encoder counter. Two buttons connected to myRIO DIO inputs will also control the operation of the system.



Pulse-width modulation

Channel-0 of Connector A, the digital signal on which we call $\overline{\text{run}}$ (the line over name denotes a logical "not," so we call this signal "not-run"), is connected to a motor driver circuit such that when $\overline{\text{run}}$ is 1 (high),[3] no voltage is applied to the motor; and when $\overline{\text{run}}$ is 0 (low), 20 V is applied. Your program will periodically alter this digital signal, applying an oscillating signal to the motor. The duty cycle (the percentage of time power is applied) is the percentage of time the channel is low.

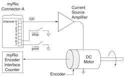


Figure L.1: a schematic of the pulse-modulation via $\overline{\text{run}}$ DIO output, the speed measurement via the FPGA encoder input, and the UI buttons $\overline{\text{print}}$ and $\overline{\text{stop}}$.

Encoder and counter

An optical encoder is mounted on the shaft of the dc motor. The encoder is the Avago HEDS-5640-A06. It is a quadrature encoder. It has 500 lines (i.e. counts per revolution, CPR), and two LED/Phototransistor pairs. The two signals (e.g. `A` and B) are 90 degrees of phase apart. If the encoder is rotating clockwise, `A` leads `B` by 90 degrees. If the encoder is rotating counter-clockwise `A` lags `B` by 90 degrees. This is how direction is encoded. In total, then, there are $4 \times 500 = 2000$ state changes per revolution. Therefore, each encoder state change corresponds to a motor rotation of $1/2000$ revolution, called a basic displacement increment (BDI).
The two phases are connected to a quadrature counter. The counter detects two state changes (one down-to-up and one up-to-down) for each line passage. Two changes for phase A and two for phase B: a total of four state changes for each line. So, for one revolution the counter totals 2000 state changes, and counts up or down depending of which phase leads.
An encoder counter in the 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.

Field Programable Gate Array

Initializing the encoder counter

---

[3] We use the C notation that the integer 1 means boolean true and the integer 0 means boolean false.

Counting of the encoder state changes is accomplished by the FPGA associated with the Xilinx Z-7010 system-on-a-chip, with dual Cortex-A9 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. The function `EncoderC_initialize`, included in the `me477` library, alters the appropriate control registers to initialize the encoder interface on Connector C. The prototype for the initialization function is:

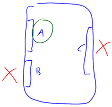
```
NiFpga_Status EncoderC_initialize(
  NiFpga_Session myrio_session,
  MyRio_Encoder *channel
);
```

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, above `main`,

```
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;
```



#include "fix Encoder.h"
~~#include "emulate.h"~~

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 on $\overline{\text{run}}$ that applies an average voltage to the motor determined by the duty cycle. The period and 1 BTI will be controlled by calling N `wait` functions, each of which takes the same deterministic amount of time. During the first M `waits` each period, voltage will be applied to the motor. See the first graph in Fig. L.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 Fig. L.2.

The algorithm should be implemented as a finite state machine (see Lec. 04.4). As shown in Fig. L.3, the machine will have five possible states: high, low, speed, stop, and (the terminal) exit. The inputs will be the `Clock` variable, and channels 6 and 7 (for the $\overline{\text{stop}}$ and $\overline{\text{print}}$ buttons). The outputs will be $\overline{\text{run}}$, `Clock` (which sometimes needs reset to `0`), and the motor speed printed to the LCD display. The corresponding state transition table, listing all possible transitions, is shown in Table L.1.

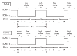


Figure L.2: $\overline{\text{run}}$ waveforms for (top) when the $\overline{\text{print}}$ button is not being pressed and (bottom)when the $\overline{\text{print}}$ button is being pressed.

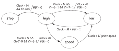

stop
high
low
speed
Clock = N && Ch-6=0 / run = 0
Clock = N && Ch-6= 1 && Ch-7= 1 / Clock =0 run = 0
Clock = M / run = 1
Clock = N && Ch-7=0 && Ch-6=1 / Clock =0 run = 0
Clock = 1/ print speed

Figure L.3: State transition diagram.

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 using `initializeSM`, described, below.
3. Request, from the user, the number (N) of wait intervals in each BTI.

Table L.1: ×: irrelevant input, ○: no change in output.

when state is	and input is			then output			and make state
	$\overline{\text{stop}}$ Ch6	$\overline{\text{print}}$ Ch7	`Clock`	$\overline{\text{run}}$ Ch0	`Clock`	speed	
high	1	1	N	0	0	○	low
high	1	0	N	0	0	○	speed
high	0	×	N	0	0	○	stop
low	×	×	M	1	○	○	high
speed	×	×	1	○	○	print	low
stop	×	×	×	○	○	○	exit

4. Request the number (M) of intervals 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

In addition to `main`, several functions will be required, as described, below. These functions include one for each state: `high` for high, `low` for low, `speed` for speed, and `stop` for stop.

`double_in` To execute the user I/O you may use the routine `double_in` developed in Lab Exercise 01, or you may simply call it from the `me477` library:

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

`initializeSM` Perform the following:

1. Initialize channels 0, 6, and 7 on Connector A, in accordance with Fig. L.1, by specifying to which register each DIO corresponds. For example, for Channel 6,[4]

```
Ch6.dir = DIOA_70DIR; // "70" used for DIO 0-7
Ch6.out = DIOA_70OUT; // "70" used for DIO 0-7
Ch6.in = DIOA_70IN;   // "70" used for DIO 0-7
Ch6.bit = 6;
```

2. ~~Additionally, initialize Connector A DIO Channels 1 and 2 in the usual way. Furthermore, set them to 1 and 0, respectively, via `Dio_WriteBit`. (An example would be `Dio_WriteBit(&Ch1, NiFpga_True);` which sets Channel 1 to 1.) This sets the motor direction via its input pins INA and INB (1, 0 is "positive" rotation and 0, 1 is "negative"). See the motor driver manual for more information.~~
3. Initialize the encoder interface. See above.
4. Stop the motor (set $\overline{\text{run}}$ to 1).
5. Set the initial state to low.
6. Set the `Clock` to 0.

`high` If `Clock` is N, set it to 0 and $\overline{\text{run}}$ to 0.
If Ch7 is 0, change the state to speed.
If Ch6 is 0, change the state to stop.
Otherwise, change the state to low.

`low` If `Clock` is M, set $\overline{\text{run}}$ to 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.

RPM

`vel` 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, set $\overline{\text{run}}$ to 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 laboratory procedure of Lec. 04.L.)

`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
```

---

[4] See `NiFpga_MyRio1900Fpga30.h` and `MyRio1900.h` for more description of the NI FPGA U8 Control `enums` that specify register addresses. For instance, `DIOA_158DIR` is short for `NiFpga_MyRio1900Fpga30_ControlU8_DIOA_158DIR`, which is at address 0x181C6, and stores the "direction" (input or output) of a DIO pin in the upper bank (pins 8–15) of Connector A.

The clock frequency of our microprocessor is 667 MHz.[5] Note carefully how the branch instructions are used. Determine the exact number of clock cycles[6] for the code to execute, accounting for all instructions. From that, calculate the delay interval in ms.

When free running, the speed of the motor is approximately 2000 RPM. Considering all the above, 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"
```

fix Encoder.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 (4-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:

floor(3.14) = 3

floor(3.8) = 3

floor(3.999) = 3

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

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 $d = -2^{31}$.

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

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: $\text{mod}(-4294967292, 2^{32}, -2^{31}) = +4$. Note that C is automatically implementing Equation 2 and this description is to deepen our understanding.

Laboratory Procedure

1. Examine the circuit on the breadboard on Connector A of the myRIO. The push button switches of Fig. L.4 connect channels 6 and 7 to ground when pressed. Note: These channels have pull-up resistors.

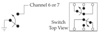


Figure L.4:

2. Use the oscilloscope to view the waveform produced by your program. For example, use $N = 5$, $M = 3$.
3. 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?
4. 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!?
5. Describe how you made this measurement and discuss any limitations in accuracy. In a later lab, we will find ways of overcoming this limitation.
6. 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;
```

---

[5] See Instruments (2013).

[6] See ARM (2012), Appendix B.

To record an accurate velocity, temporarily comment-out the `printf_lcd` statement in `speed`, and hold down the `Ch7` switch while you start the program.

7. 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 Resource 9 for more details.

   The Matlab file must be called `Lab4.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.

   For your report, the array can be plotted using the Matlab `plot` command.

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

8. 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 modifying the state transition diagram correct this problem? How would you modify the state transition table? Modify your program to correct the $M = 1$ case. Test the result.

A better way to PWM

In this exercise, among other things, we have come to understand PWM and the limitations of implementing it in the way we have. Fortunately, there is a better way: using the PWM capabilities from the FPGA, accessible via `PWM.h`.

The PWM example (`myRIO Example - PWM`) from the NI archive `C_Support_for_myRIO_v3.0.zip` shows how to do this. This method mitigates several of the issues encountered in this exercise, especially those related to duty-cycle resolution and "jerky" operation due to low PWM rates. The FPGA-based PWM can operate as high as 10 MHz, but, as with our finite state machine implementation, loses duty cycle resolution as its rate increases. So, although we have a higher rate, and therefore more cushion, the same issue of balancing PWM frequency and duty cycle resolution remain.

Of course, we still might need a finite state machine for controlling the state at a higher-level. For instance, we might include a knob for controlling the duty cycle of the FPGA PWM. This and the encoder, speed, and stop functionality of the finite state machine of the exercise could have a much lower frequency (governed by `wait`) than the PWM. Decoupling the timing for processes at much different rates, like this, is typically advantageous.

Resource R9 Saving myRIO C data to a Matlab file

The following C functions[7] write data of types `double` or `char` to a Matlab `.mat` file. They are included in the `me477` library. Be sure to `#include "matlabfiles.h"`.

Use the following functions to open a named file on the myRIO, and successively add any number of data arrays, variables, and strings to the file. Finally, close the file.

Open a `.mat` file The prototype for the open function is

```
MATFILE *openmatfile(char *fname, int *err);
```

where `fname` is the filename, and `err` receives any error code. The function returns a structure for containing the Matlab file pointer.

A typical call might be:

```
mf = openmatfile("Lab.mat", &err);
if(!mf) printf("Can't open mat file %d\n", err);
```

For this course, always use the file name: `Lab.mat`. Notice the use of pointers.

Add a matrix The prototype of the function for adding a matrix to the Matlab file is

```
int matfile_addmatrix(
  MATFILE *mf,
  char *name,
  double *data,
  int m,
  int n,
  int transpose
);
```

---

[7] See `http://www.malcolmmclean.site11.com/www/MatlabFiles/matfiles.html`.

where `mf` is the Matlab file pointer from the open statement, `name` is a `char` string containing the name that the matrix will be given in Matlab, `data` is a C data array of type `double`, `m` and `n` are the array dimensions, `transpose` takes value of 0 or 1 to indicate where the matrix is to be transposed.
For example, to add a 1-D matrix the call might be

```
matfile_addmatrix(mf, "vel", buffer, IMAX, 1, 0);
```

Or, to add a single variable the call might be

```
double Npar;
Npar = (double) N;
matfile_addmatrix(mf, "N", &Npar, 1, 1, 0);
```

Again, note the use of pointers, and the cast to `double`.

Add a string The prototype of the function for adding a string to the Matlab file is

```
int
matfile_addstring(
  MATFILE *mf,
  char *name,
  char *str
);
```

where `mf` is the Matlab file pointer from the open statement, `name` is a `char` string containing the name that the matrix will be given in Matlab, and `str` is the string.
For example, to add a string the call might be

```
matfile_addstring(mf,"myName","Bob Smith");
```

Close the file After all data have been added, the file must be closed. The prototype of the function for closing the Matlab file is

```
int matfile_close(MATFILE *mf);
```

where `mf` is the Matlab file pointer from the open statement.
For example, to close the Matlab file the call might be

```
matfile_close(mf);
```

Example code Putting these ideas together:

```
mf = openmatfile("Lab.mat", &err);
if(!mf) printf("Can't open mat file %d\n", err);
matfile_addstring(mf, "myName",  "Bob Smith");
matfile_addmatrix(mf, "N", &Npar, 1, 1, 0);
matfile_addmatrix(mf, "M", &Mpar, 1, 1, 0);
matfile_addmatrix(mf, "vel", buffer, IMAX, 1, 0);
matfile_close(mf);
```

Transfer file to Matlab After the `Lab.mat` file has been created, it can be transferred directly to Matlab.

1. In Eclipse's right pane of the Remote Systems Explorer perspective, select `172.22.11.2`, and select the icon Refresh information of selected resource.
2. Double click on the Matlab data file: `172.22.11.2>SftpFiles>MyHome>Lab.mat`.
3. The `Lab.mat` file will be opened in Matlab on your laptop. Use Matlab's `whos` command to list all of the named variables in the workspace.
4. In Matlab navigate to a convenient folder on your laptop. Then, issue the `save('Lab.mat')` command to save the Matlab workspace, locally. The file can later be opened from a Matlab script, using the command `load('Lab.mat')`, for plotting or analysis.

Note: You will also find the `Lab.mat` file in the `RemoteSystemsTempFiles` folder within your workspace folder.

Resource R10 Copley 412 analog amplifier setup

This should be adequate (and safe) for the Clifton Precision JDH-2000-V-1C or similar dc motor. It has a stall voltage of 24 V and stall current of 2.18 A.

Resistor settings

- RH15 Peak Current 6.2 kΩ (20 % of 20 A = 4 A)
- RH16 Continuous Current Limit 0 Ω (16 % of 20 A = 3.2 A)
- RH17 Peak Current Time Limit open (1 second)
- RH20 Armature Inductance 49.9 kΩ (0.6 to 1.9 mH)

Capacitor settings

- CH18 Armature Inductance 4.7 nF (0.6 to 1.9 mH)

Dip switch settings

- S1 Ground-active Enable `OFF` (up, away from the board)
- S2 Torque Mode `ON` (down, toward the board)

Gain adjustment

We will be operating the amplifier in `TORQUE MODE`. For transconductance, (output current / input voltage) = peak current / 10 V, which can be set up with the following steps:

1. Set S2 `ON`.
2. Set Ref Gain fully CW.
3. Set Loop Gain fully CCW.
4. Adjust the transconductance gain to 4 A / 10 V.
    a) To increase gain, turn Loop Gain CW.
    b) To decrease gain, turn Ref Gain CCW.

05

Threads and interrupts