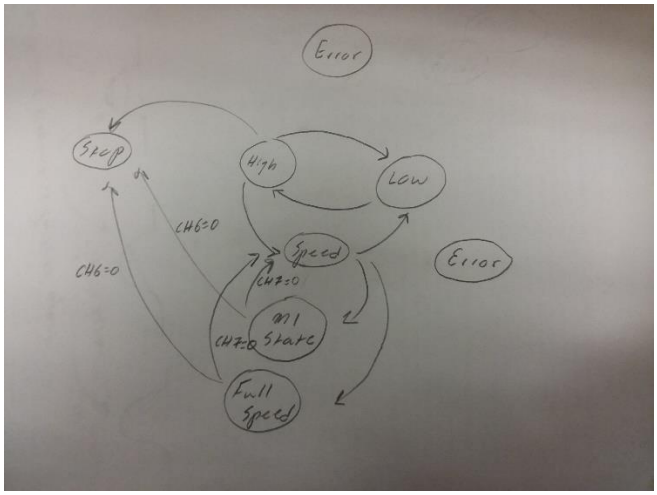
RILEY POOLE
ME 477 – LAB REPORT 04
WINTER 2019

I. DESCRIPTION
II. TESTING
III. RESULTS
IV. SPECIFIC QUESTIONS IN THE ASSIGNMENT
V. FUNCTIONS OF THE PROGRAM
VII. MATLAB CODE
IIX. ALGORITHMS AND PSEUDOCODE

I. DESCRIPTION

The primary purpose of this program is to setup and test a state machine. It will be used primarily to control the speed of a DC motor with a PWM signal and has four states defined by the lab handout and three user created states. Two to control the PWM signal, one to calculate the current speed, and one to stop the machine. I also added three states FULL_SPEED, M1_STATE, and ERROR. The program will accept inputs on the keypad from the user to set the duty cycle (and therefore the speed) of the motor. It will display the velocity of the motor in RPM on the LCD display and save the results to a Matlab file.

Since changes to the states have been made the new diagram is shown below. The output states have been omitted.



THE MAJOR TASKS PERFORMED BY THIS PROGRAM WERE

- Request from the user the duty cycle in terms of integer values of N (number of wait cycles in a period) and M (number of wait cycles the motor is on every period)
- Set the speed of the motor with a PWM signal controlled with the N and M (duty cycle)
- Measure the speed (RPM) of the motor and save the result to a Matlab file
- Stop the motor when commanded

THE LIMITATIONS OF THIS PROGRAMS CAPABILITY ARE

- the wait function is not well implemented
    - a while loop is a poor timer
- motor uses an incremental encoder which will may more undetected errors as compared to an absolute encoder
- duty cycle cannot be entered manually as a percentage
- no runaway protection
    - if the motor exceeds its set speed there is no control to stop it
- feedback is not used
    - although the speed is calculated the motor is an open loop system where the PWM signal is set rather than checking the speed of the motor
- motor direction is not controlled

EXPLAIN ANY ALGORITHMS

The control is implemented as a state machine with seven components. Each state leads to others based on the status of the program. Some of the states will call other user defined functions such as vel(). Vel() uses the current and previous encoder counts to estimate the speed as a floating point value. Timing in the program is accomplished with a wait() function that simply counts down from a large value.

FULL_SPEED is when the values of N and M are the same indicating a duty cycle of 100%. The only way this is achievable is by setting the bit 0 low for the whole program.

M1_STATE is when the M value is one. This state is designed to prevent SPEED from causing a runaway condition. ERROR is a special state that is only accessible to main() when main detects that “Clock” has exceeded the period by more than 2 counts and could potentially cause the motor to spin uncontrolled.

HIERARCHICAL STRUCTURE

- main()
  - MyRio_Open
  - MyRio_IsNotSuccess
  - EncoderC_initialize
  - initializeSM
    - EncoderC_initialize
    - printf
    - Dio_WriteBit
  - request_N_and_M
    - printf_lcd
    - fgets_keypad
    - sscanf
  - state_table[curr_state]
    - high
    - low
    - speed
    - stop
    - full_speed
    - m1_state
    - error
  - getkey
  - MyRio_Close
- high
  - Dio_ReadBit
  - Dio_WriteBit
- low
  - Dio_Writebit
- speed
  - vel
  - printf_lcd
- stop
  - Dio_WriteBit
  - printf_lcd
  - openmatfile
  - printf
  - matfile_addmatrix
  - matfile_addstring
- m1_state
  - Dio_WriteBit
  - Dio_ReadBit
- full_speed
  - Dio_ReadBit
- error
  - Dio_WriteBit
  - printf

II. TESTING

1. Test the program can accept integer values of N and M, set the duty cycle, and return the speed to the LCD screen
    - **a.** **Check #define mode is 0;** compile and run the program
    - **b.** Enter <50><Ent>
    - **c.** Enter <40><Ent>
    - **d.** Duty cycle is now set to 80% check oscilloscope says 20% duty cycle which is the inverse (because bit zero is on when current is off)
    - **e.** Press CH7
        - i. there should be a tachometer reading displayed on the LCD and should read roughly 2000 RPM
    - **f.** Press CH6 to shut down the motor and end the program
2. Test the program can interpret an M=1 condition and react
    - **a.** run the program
    - **b.** Enter <10><Ent>
    - **c.** Enter <1><Ent>
    - **d.** Duty cycle is now 10%
        - i. Check oscilloscope says 90% duty cycle
        - ii. Press CH7
            1. there should be a tachometer reading displayed on the LCD
               Note: a limitation of this machine is that it can not move with very low duty cycle <40% and speed button slows it down for low M values
    - **e.** Press CH6 to shut down the motor and end the program
3. Test the program can interpret an M=N condition and react
    - **a.** run the program
    - **b.** Enter <50><Ent>
    - **c.** Enter <50><Ent>
    - **d.** Duty cycle is now 100%
        - i. Check the oscilloscope is a flat DC line near 0V or very close
    - **e.** Press CH7
        - i. there should be a tachometer reading displayed on the LCD and should read roughly 2100 RPM
    - **f.** Press CH6 to shut down the motor and end the program
4. Test that the program will return a Matlab file of speeds
    - **a.** Hold down CH7 for this entire process
    - **b.** Set #define mode 1, compile and run
    - **c.** Enter <25><ENT>
    - **d.** Enter <25><ENT>
    - **e.** Duty cycle is now 100%
    - **f.** Release CH7 and quickly press CH6 to save the results
    - **g.** Check there should be a new file “Lab.mat” that can be used to plot the results

III. RESULTS

HOW SUCCESSFULLY THE PROGRAM RUNS AND UNSOLVED PROBLEMS

I feel the program has reached its goals. Right now the program can handle integer values of M and N and the special cases where M=N and M=1.

One unsolved problem is the efficiency of the program. In order to handle the special cases I had to add comparisons and flags that go in functions critical to speed like vel(). The printf() delay in speed is also left unresolved and is likely not possible to fix without using another thread. Even with the printf() function commented out SPEED still causes a 5 ms delay. I feel there is one extra clock cycle hiding between the transition from SPEED to LOW but I cannot find it. I tried setting the increments of the clock to the inside of the functions, but this did not explain the behavior. The debugger runs with CH7 depressed just fine, but while running the program with low values of N, CH7 will cause a permanent current-on state. The accuracy of the software tachometer also degrades for low values of N and for M = 1. The tachometer in general has a poor accuracy and almost always reads high. I have not been able to trace this error and believe it has to do with calculating the value of BTI = N * wait() period.

IV. SPECIFIC QUESTIONS IN THE ASSIGNMENT

Calculate the delay interval in milliseconds for wait()

	Cycles Per Instruction	# Times Called	Total Cycles	#
wait+0 push {r11}	2	1	2	1
wait+4 add r11, sp, #0	1	1	2	2
wait+8 sub sp, sp, #12	1	1	1	3
wait+12 mov r3, #417000	1	1	1	4
wait+16 str r3, [r11, #-8]	2	1	2	5
wait+20 b 0x8ed4 <wait+36>	1	1	1	6
wait+24 ldr r3, [r11, #-8]	2	417000	834000	7
wait+28 sub r3, r3, #1	1	417000	417000	8
wait+32 str r3, [r11, #-8]	2	417000	834000	9
wait+36 ldr r3, [r11, #-8]	2	417000	834000	10
wait+40 cmp r3, #0	1	417000	417000	11
wait+44 bne 0x8ec8 <wait+24>	0	417000	0	12
wait+48 nop ; (mov r0, r0)	1	1	1	13
wait+52 add sp, r11, #0	1	1	2	14
wait+56 ldmfd sp!, {r11}	2	1	2	15
wait+60 bx lr	1	1	1	16

TOTAL CYCLES	3336015
FREQUENCY (hz)	6.67E+08
PERIOD (s)	1.50E-09
TOTAL TIME (ms)	**5**

What inaccuracies or programming difficulties are there in using a delay routine for control and time measurement

Measuring runtime with assembly language can be difficult because a CPU is so complex. Only broad generalizations about the number of clock cycles will be able to be made based on the fundamental circuitry of the chip. The real value will depend on runtime specific factors like out-of-order processing and the relationship between control of both cores. However, even if there were a reliable way to count the number of clock cycles it would not take away the fact that this is a very cumbersome process to do every time you want to make a delay function. This is especially the case where the code is not written by hand but compiled. Each compiler will generate slightly different assembly and would require a different count down period for each.

Determine a reasonable value for N (number of delay intervals in a BTI)

1500 rev/min * 1min/60sec * 2000 counts/rev = 50,000 counts/sec
12 bits is 4096 counts
50,000 counts/sec * 1/4096 BDI/count = 12.2 BDI/sec

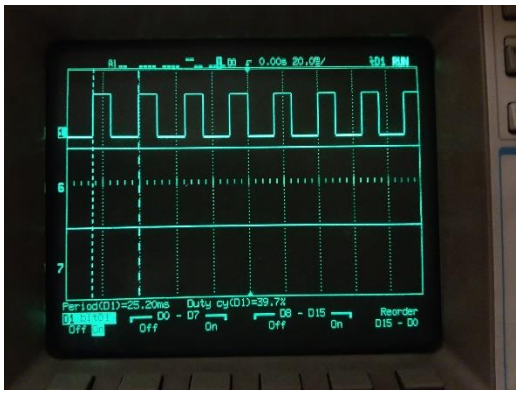
If each BDI is to capture no more than 4096 counts
Will need at least 13 BDI/sec to capture them
∴1 BTI = 1/13 sec = 76.9 ms

If each N takes 5 ms then
76.9 ms/BTI * 1/5 N/ms = 15.38 N/BTI

∴We will need at least N = 16

However, experimentally I found this value to be much higher roughly 50 for good results.

2. Use the oscilloscope to view the waveform produced by your program. For example, use N = 5, M = 3



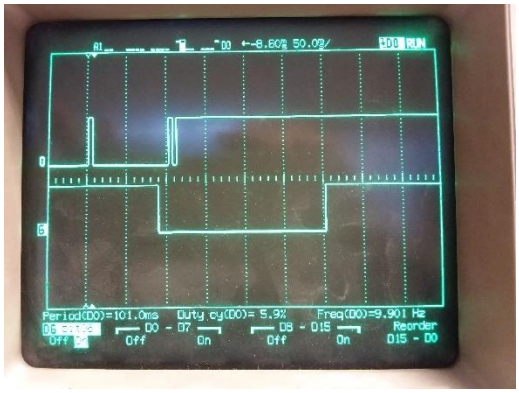
3. Compare your program's printed output (on the LCD display) to the steady state motor speed as measured with the optical tachometer.

With M = 50 and N = 50 my software tachometer measures: 2138 RPM
The optical tachometer measures: 1971 RPM
The experimental error is: +8.4%

4. Use the oscilloscope to view the start/stop waveform produced by our program, and to measure the actual length of a BTI. Is it what you expect? If not, why not?

View the start/stop waveform produced by our program:

M = 100 N = 95 Duty Cycle 95%



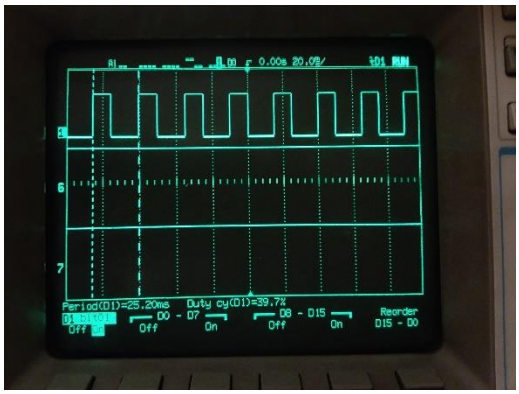
Measure the actual length of a BTI. Is it what you expect? If not, why not?:

I set N to 20 (M = 10) so I would expect a BTI of 100ms. The oscilloscope says that the period is 101ms.
This is probably due to all of the tasks that have to be accomplished in the I/O functions Dio_Writebit() and Dio_Readbit(). These I/O functions are called throughout high() and low() which would add error to the PWM signal. The wait() function is also not very accurate as it is controlled with a countdown loop and there may be some slight discrepancies. Lastly, the clock counter itself should be considered. The clock doesn't reset until high() tells it too. So "Clock" hangs with a value greater than the set value for a tiny fraction of a second.

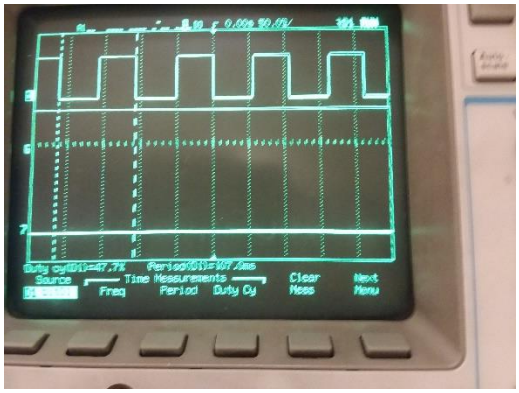
5. Repeat the previous step while printing the speed (press the switch). what does the oscilloscope show has happened to the length of the BTI? What's going on!?

The BTI length increases dramatically from 101ms to 107ms. The LOW state (current off) also increases its span. Running through the debugger I noticed that speed() calls a printf_lcd() function which operates at serial speeds which is much slower than the processor. Printing to the LCD screen takes considerable time leaving the machine in the LOW state longer and speeding up the motor. Commenting out the printf_lcd() command worked to keep the LOW (current on) cycle from increasing.

Before Pressing the Switch:



After Pressing the Switch



6. Describe how you made this measurement and discuss any limitations in accuracy.

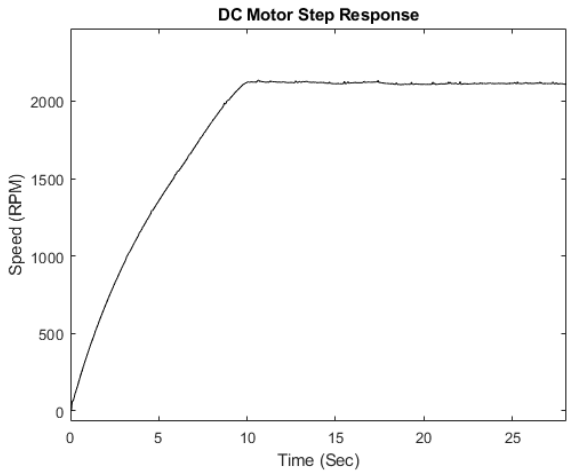
The time taken to print to the LCD is approximately 8ms.

For N=20 M = 15 Duty Cycle = 75% the delay is 8 ms
For N=100 M = 75 Duty Cycle = 75% the delay is 8 ms
For N = 200 M = 150 Duty Cycle = 75% the delay is 10 ms

To make these measurements I measured the period before and after pressing the “speed” button. The limitation in measurement is the accuracy of the period measurements the oscilloscope can make.

7. Plot of speed results

Steady State: 2114 rpm
Time Constant: 4.3 seconds (2%-first order)



The steady state value was determined by by averaging the values were the response had plateaued.

The time constant is estimated by assuming the motor has a first order response and finding the time to reach 98% of the settling time. Then using the relation

rise_time = 2.2 * time_constant

EXTRA. Fixing M = 1

The problem is that the state “LOW” needs the clock to equal the value of M before it can switch back to a high (current off) state. The “SPEED” state needs a clock input of 1 to advance to low which means the clock is then 2. It is never possible to get out of “LOW” and the motor runs away. This could be alleviated with substantial changes to the functions that are already available or by defining a new state. The new state will be called “M1_STATE” it will make sure that low is called once and high is called N-1 times. It will allow the use of “STOP” and “SPEED” and will have a global flag to allow “SPEED” to send it back to “M1_STATE”.

POSSIBLE IMPROVEMENTS AND EXTENSIONS

- A better way to delay would be to use an interrupt timer
- A feedback algorithm could be implemented to maintain the speed at a set level
- The user could be able to enter the duty cycle directly
- Adding an H-bridge to control the direction of the motor

V. FUNCTIONS OF THE PROGRAM

- **main()**
  - Prototype:
    - int main(void)
  - Purpose:
    - test the state machine by setting various duty cycles
  - Rationale for creation:
    - Need a procedural section to layout testing environment
  - Inputs and parameters:
    - No inputs or parameters used
  - Return:
    - Sets the motor speed
    - Saves motor response to matlab file
    - Prints the motor speed to the LCD
    - Returns errors if initialization fails
- **initializeSM()**
  - Prototype:
    - void initializeSM(void)
  - Purpose:
    - set the control registers to an initialization state
    - prevent the motor from running, set the initial state to low 0, and reset the clock
  - Rationale for creation:
    - convenient way to set up the state machine to begin
  - Inputs and Parameters:
    - None – procedural
  - Returns:
    - controls registers set for ch0 ch6 ch7
    - error during encoder initialization
    - Motor set to run (ch0 to 1)
    - current state to low
    - clock set to 0
- **high()**
  - Prototype:
    - void high(void)
  - Purpose:
    - sets the PWM signal high
  - Rationale for creation:
    - convenient way to control the PWM signal
  - Inputs and parameters:
    - No inputs or parameters used
  - Return:
    - sets run to 0 (on) and current flows through the motor
    - sets next state appropriately

- **low()**
  - Prototype:
    - void low(void)
  - Purpose:
    - sets the PWM signal low
  - Rationale for creation:
    - convenient way to control the PWM signal
  - Inputs and parameters:
    - none
  - Return:
    - sets run to 1 (off)
    - sets net state to “HIGH”
- **vel()**
  - Prototype:
    - double vel(void)
  - Purpose:
    - read the encoder and calculate the speed in BDI/BTI
  - Rationale for creation:
    - convenient way to measure the speed of the motor with the encoder
  - Inputs and parameters:
    - None
  - Return:
    - speed in BDI/BTI
- **stop()**
  - Prototype:
    - void stop(void)
  - Purpose:
    - stop the motor
    - set next state to “EXIT”
    - save the responses to matlab
  - Rationale for creation:
    - convenient way to stop the motor and save the results
  - Inputs and parameters:
    - None
  - Returns:
    - sets run to 1 (off)
    - sends stopping message to LCD screen
    - sets the next state to “EXIT”
    - save the responses to matlab

- **speed()**
  - Prototype:
    - void speed(void)
  - Purpose:
    - calls vel() to read speed then converts to RPM
    - prints the current speed to the LCD screen
    - save the results for use in matlab
  - Rationale for creation:
    - convenient way to check the speed and save the results
  - Inputs and parameters:
    - None
  - Returns:
    - speed in RPM
    - saves the speed for use in matlab
    - prints the speed to the lcd screen
    - sets the next state to “LOW”
- **request_N_and_M()**
  - Prototype:
    - void request_N_and_M(void)
  - Purpose:
    - get the duty cycle from the user through the keypad
  - Rationale for creation:
    - convenient way to get the duty cycle with N and M
    - allows for setting the duty cycle within the while loop in main easier
  - Inputs and parameters:
    - None
  - Returns:
    - collects M and N values
    - returns the set duty cycle to the LCD screen
- **wait()**
  - prototype:
    - void wait(void)
  - purpose:
    - waits for a period of time
  - parameters
    - none
  - returns
    - none
- **m1_state()**
  - prototype:
    - void m1_state(void)
  - purpose:
    - React to an M = 1 condition
  - parameters
    - none
  - returns
    - none

- **full_speed()**
  - prototype:
    - void fullspeed(void)
  - purpose:
    - React to an M = 1 condition
  - parameters
    - none
  - returns
    - none
  - prototype:
    - void error(void)
  - purpose:
    - Called by main to halt the program
  - parameters
    - none
  - returns
    - none

VII. MATLAB CODE

```
close all; clc

wait_interval = 5e-3;
period = N*wait_interval;
time = [];
for i = 1:length(Speed)
    time(i) = period*i;
end

plot(time,Speed,'k'); xlabel('Time (S)'); ylabel('Speed (RPM)');
title('DC Motor Step Response')

%find steady state
steady_state = mean(Speed(300:500));

%find the time constant
for i = 1:length(Speed)
    if(Speed(i) > 0.98*steady_state)
        break
    end
end

time_constant = time(i)/2.2
```

IIX. ALGORITHMS AND PSEUDOCODE

```
main()
      open the myRIO session
      intialize the state machine
            initialize the encoder EncoderC_initialize()
            initializeSM()
      request starting values from the user for N and M
      start the state machine loop and don't stop until the curr_state is
"EXIT"
      close the myRIO when "EXIT" is asserted

initializeSM()
      initialize channels 0,6,7
      initizlize the encoder interface
      stop the motor
      set the initial state to "LOW"
      check for M1 condition
      check for full_speed condition
      set the clock to 0

high()
      if the clock is N
            set clock to 0
            set run to 0
            if ch 7 is 0
                  change the state to "SPEED"
            if ch 6 is 0
                  change the state to "STOP"
            otherwise change the state to "LOW"

low()
      if clock is M
            set run = 1
            change state to high

speed()
      use vel() to read counter and compute the speed in BDI /BTI
      convert the speed to RPM
      change the state to "LOW"

vel()
      read the current encoder count
      compute the speed as the difference between the current and previous
counts
      replace the previous count with the current count for use in the next
BTI
      Return the speed (double) to the calling function
      print the speed in units of RPM

stop()
```

```
      stop the motor
            set run = 1
      clear the LCD and print "STOPPING"
      set the current state to "EXIT"
      save the response to a matlab file

request_N_and_M()
      request the user enter on the keypad the number of N wait intervales in
each BTI
      request the user enter on the keypad the number of M intervals that the
motor is on in each BTI
      give the user feedback

wait()
      count down from a really high number to waste time

m1_state()
      assert that m1_state has been called
      read channel 6 and 7
      for every N passes do one low state
      check for stop and print buttons

full_speed()
      assert full_speed
      read channel 6 and 7
      set clock to 0 just in case

error()
      turn bit one to current off
      set the current state to stop
```