ECE341

**Lab10 - Input Capture to Measure Motor Speed**

Report

**Seth Cram**

**12/07/2021**

**Introduction:**

Goal:

Use the input capture module to measure frequency, which will allow us to find the speed of the DC motor.

Background Information:

The DC motor has hall effect sensors that generate a signal in response to a magnetic field. This magnetic field is created through the rotation of two magnets in and out of the range of hall effect sensors SA and SB. Because of the sensor's positioning, SB lags behind SA by about 1 digital output. The result is called Quadrature Encoding, and it's useful to determine the shaft speed and determine what direction the DC motor is turning in. For this lab, we’ll only use SB since we’ll only be determining the motor’s speed and SA is not a designated input capture pin.

To determine the speed of the DC motor, we’ll have to capture a timer’s value when an edge is generated by SB. Since the duty cycle generated by SB isn’t perfect due to hall effect tolerance, we should measure the rising edge or the falling edge, not both. The input capture module will capture a specified edge for us and store a timer’s value at the time of the event. We can just read this value out of the FIFO memory that the input capture module stores the timer’s value in.

We could poll until something is stored in the FIFO memory area, or we could configure the input capture module to generate an interrupt every edge we told it to capture. For this lab, we’ll use the falling edge and configure the input capture module ISR every interrupt it generates. Now that we have the time between consecutive falling edges generated by SB, we can invert this to get a frequency. This frequency is the motor’s rotations per second, its speed, since a sensor only pulses once every revolution.

If desired, we could configure the input capture module to only capture every 4th or 16th edge using a prescaler. We'll only use a 16-bit timer, timer 3, since combining two 16-bit timers to get a 32-bit timer would only be appropriate for incredibly widely spaced out events. In order to keep track of higher counts, we could also use a timer’s prescaler, but doing this decreases our resolution. As always, when dealing with timers we want to clock them as fast as possible without overflow.

Similar to lab 9, we will once again be using the H-bridge to communicate between the PIC32 and DC Motor/tachometer. Sensor A and sensor B are outputs from the H-bridge to the PIC32. The DC motor tachometer is also used in this lab to measure the analog speed of the motor. It acts as a voltage to frequency converter for the DC motor.

Some important parameters for verification in the lab is that using a 10V supply, the max motor speed is 525 RPS, and due to friction the min motor speed is around 79.5 RPS. We’ll be using input capture interrupt 5, since SB is connected to input capture pin 5.

Plan:

First, I plan on building lab 10 off of lab 9. So, all function code for lab 10 will be included in a new file that’ll be included by the main() file used in lab 9. In this new file I’ll create an input capture initialization function that’ll set sensors A and B as inputs as well as clear, configure, and open the input capture 5 module. Most of this will be done using macros provided by the lab handout. We’ll use two timers during this lab, since the period registers of the timer’s are set differently to fulfill different design goals.

Then, I’ll create the timer 3 initialization function based off of the timer 2 one present in lab 9. Its priority and sub-group priority is specified in lab, and it’ll be opened with a prescaler of 256 as well as a period register of the maximum value possible. We’ll be setting the period register for timer 3 to its maximum value of 65,536 to avoid incorrect period measurement when the timer rolls-over multiple times and to achieve a high level of resolution. This interrupts corresponding ISR will merely toggle LEDC and clear the timer 3 flag.

Finally, the Input Capture ISR will be structure based off-of Dr. J’s pseudocode provided in-class and the code given in the lab handout. Also, truncating our unsigned integers to unsigned shorts allows for period calculation to be correct when the timer rolls over a single time. I’ll store each calculated t\_diff as an element in an array, and the counting variable will reset to zero every time it reaches 16, which will make the array act like a circular buffer. We will average multiple period measurements to reduce measurement noise. Once 16 or more t\_diffs are recorded, I’ll calculate the speed through averaging these, multiplying that by the timer prescaler over the peripheral bus clock frequency, and inverting that to get the frequency. I’ll need to use floats for this to avoid large rounding errors. The speed will be stored in a global variable. Once rps is calculated, I’ll output the appropriate message to the LCD

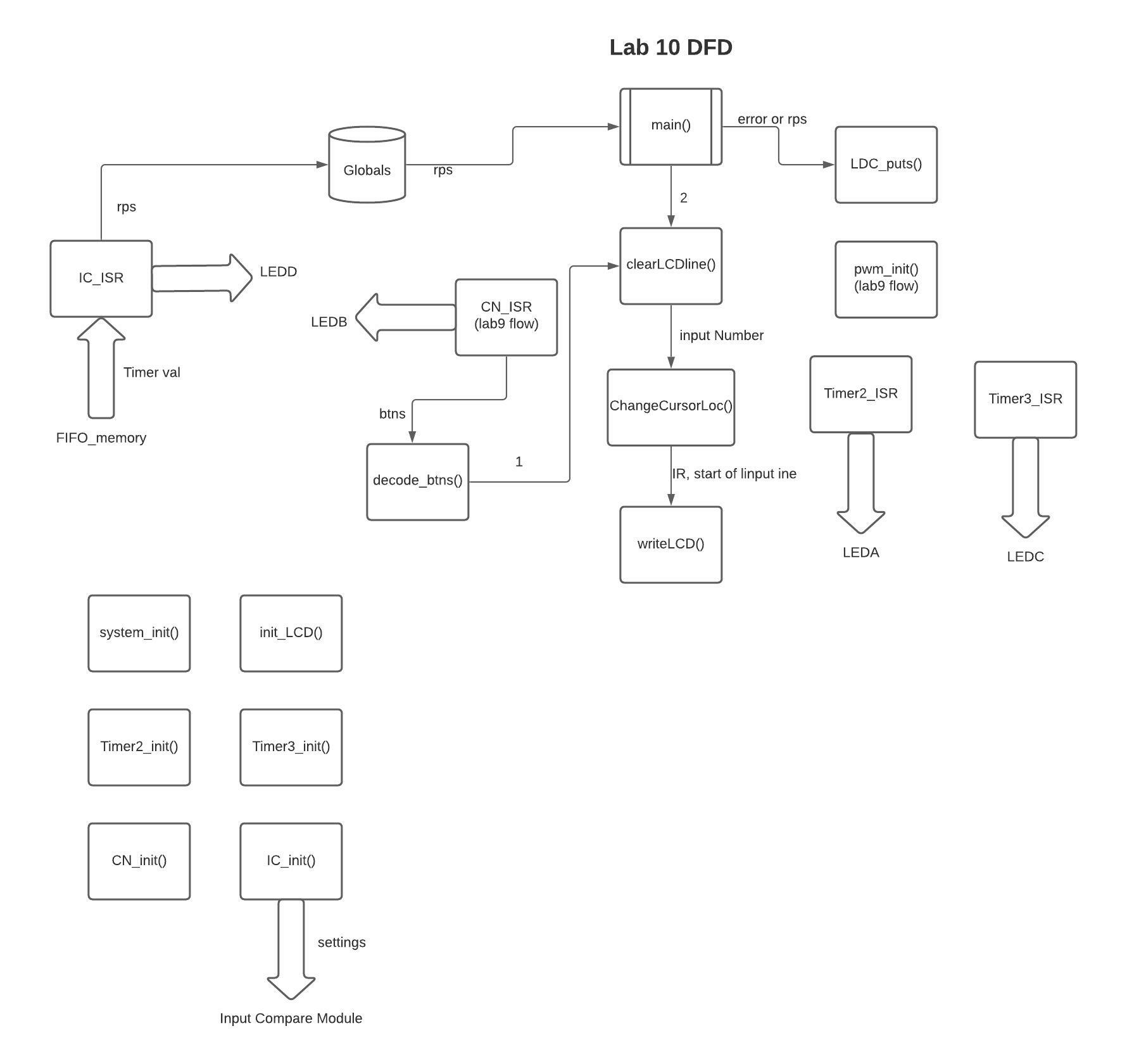
To only write the second line of the LCD in the input capture ISR, and only the first line of the LCD in the Change Notice ISR, I’ll need to create a new LCD function to clear any desired line of the LCD and put the cursor at the start of that line when done. Delving into my LCDlib file, I’ll create this by accepting what line to clear as an argument to a function and then writing 16 blank spaces to the specified line, then changing the cursor’s location back to the start of that line. I’ll have to take into account using the busy flag to see if the LCD Controller is ready yet when changing the cursor location. Like in previous labs, everytime I write to the LCD, I’ll also have to disable, and after writing, re-enable the change notice interrupt.

Back to the main() file, I’ll do everything in lab9, then also initialize the input capture module, and Timer3 as an interrupt. Within the while(1) loop, I’ll disable the Change Notice interrupt before clearing LCD line 2, formatting my desired RPS string using sprintf(), and output that formatted string onto the LCD. Finally, I’ll re-enable the Change Notice interrupt and delay for 100 ms before repeating. I’ll be sure to include the global variable in my main() file using extern.

*Background information came from lecture notes and the lab 10 handout.*

**Implementation Discussion:**

Before implementation, I designed a data flow diagram to get a visual of what files I’d need to incorporate and their overarching functioning in the grand scheme.



As seen above, we’ll need to use all the components of lab 9. Not shown is the flow of data for the unchanged lab 9 functions and ISRs. I made sure to illustrate which function/ISR was outputting to which LED, since this is a large project and getting them mixed up would be easy. The two new LCD functions we’ll need to write are also displayed above, as well as the flow of the global revolutions per second.

**Listing 1. test\_proj10.c:**

We’ll delve into the test file, look at the newly created input\_capture header and source files, and finish by looking at the changes made in the LCD library and pwm files. Much of project 10’s test file is the same as project 9’s. Differences include: the inclusion of the input\_capture and string header files, initialization of timer 3 and the input capture module, and inclusion of outputting the externed rps to the LCD in the while(1) loop.

*#include "input\_capture.h"*

*#include <string.h> //for sprintf()*

*extern float rps;*

*ic\_init(); //init input capture module*

*t3\_init();*

*while(1)*

*{*

*//clear line 2 and change cursor pos to start of line 2*

*ClearLCDline( 2 );*

*sprintf( rpsStr, "RPS = %.2f", rps ); //format output str*

*//output formatted string:*

*LCD\_puts( rpsStr );*

*//delay till next output:*

*LCD\_delay( msgDelay );*

*}*

Not shown above are the disabling and enabling of the change notice interrupt before and after interacting with the LCD.

**Listing 2. input\_capture.h:**

Before bogging ourselves down with the details of the input\_capture.c, we’ll look at its header file.

*void system\_init(void);*

*int ic\_init();*

*void t3\_init(void);*

*//global var for motor speed:*

*extern float rsp;*

This is a short list of prototypes and the inclusion of a single global variable, so it may not have warranted a full-blown separate header file. The naming for each function is fairly self-explanatory.

**Listing 3. input\_capture.c:**

Getting into the details of how we operated the input capture module, we’ll look at the internals of the above seen functions. First, system\_init() performed similarly to the pwm one, but we also had to set up the direction output pin for RD1.

Then, for ic\_init(), we set the hall effect sensors to digital inputs, opened the input capture module, and configured its interrupt.

*const unsigned int MTR\_SA = BIT\_9;*

*const unsigned int MTR\_SB = BIT\_10;*

*//Hall effect sensors SA (RD3) and SB (RD12) as inputs:*

*PORTSetPinsDigitalIn( IOPORT\_D, (MTR\_SA | MTR\_SB) );*

*OpenCapture5( IC\_ON | IC\_IDLE\_STOP | IC\_FEDGE\_FALL | IC\_CAP\_16BIT*

*| IC\_TIMER3\_SRC | IC\_INT\_1CAPTURE*

*| IC\_EVERY\_FALL\_EDGE);*

*ConfigIntCapture5( IC\_INT\_ON | IC\_INT\_PRIOR\_3 | IC\_INT\_SUB\_PRIOR\_0 );*

I will explain toward the end of the implementation discussion why I configured the input capture module with these particular settings.

The output capture ISR took the most thought. First, I initialized a combination of variables provided to me by the lab handout and my own. The lab handout code was used to read from the memory fifo, store the new time, and calculate the period. My variables and thought went into toggling the LEDD, storing the periods in an array, calculating the average and rps of the array of 16 values, and resetting the index to result in a circular array.

*//seth's vars:*

*static unsigned int captureNum = 0; //cnting var for curr capture*

*static unsigned int index = 0; //cnting var for array*

*static const int minCaps = 16; //min caps to start averaging*

*static unsigned short int arrTdiff[ 16 ]; //array to store periods (minCaps)*

*LATBINV = LEDD; //toggle LEDD*

*//store t\_diff in static array (circular buffer):*

*arrTdiff[ index ] = t\_diff;*

*if( captureNum >= minCaps - 1 ) //file 15 vals*

*{*

*int sum = 0; //sum of T\_diffs*

*int i; //for-loop cnting var*

*//loop thru array:*

*for( i = 0; i < minCaps; i++)*

*{*

*sum += arrTdiff[ i ]; //sum all Tdiff's*

*}*

*float avg = ((float) sum) / ( (float) minCaps); //avg Tdiff, curr in seconds/tick*

*//convert avg ticks/s to avg s / ticks:*

*avg = avg \* (256.0 / 10000000.0); // (ticks/s) \* (s/tick / ticks/s) = s / ticks*

*rps = 1.0 / avg; // ticks/s*

*}*

*index++; //incr array index*

*//if index maxed out:*

*if( index >= minCaps ) //fill 15 then rollover*

*index = 0; //reset array index*

*captureNum++; //a capture taken*

Not shown above is the code provided to us by Dr. J and the lab handout that was implemented in-between my variables and the first line of my actual implementation code. I also cleared the input capture interrupt flag at the end of the ISR. Although this isn’t the most efficient method since we recalculate a new average every new time capture, the increased latency made little difference to the human eye.

The timer 3 ISR and its initialization function function in the same fashion as its timer 2 counterpart from the pwm code. Although, when opening timer 3 we gave it a prescaler of 256 (the longest possible) and gave its PR also the maximum it could hold. This is because we wanted timer 3 to interrupt as little as possible. The timer 3 sub priority also was higher than the timer 2’s by one, so if they collide timer 3 will take precedence.

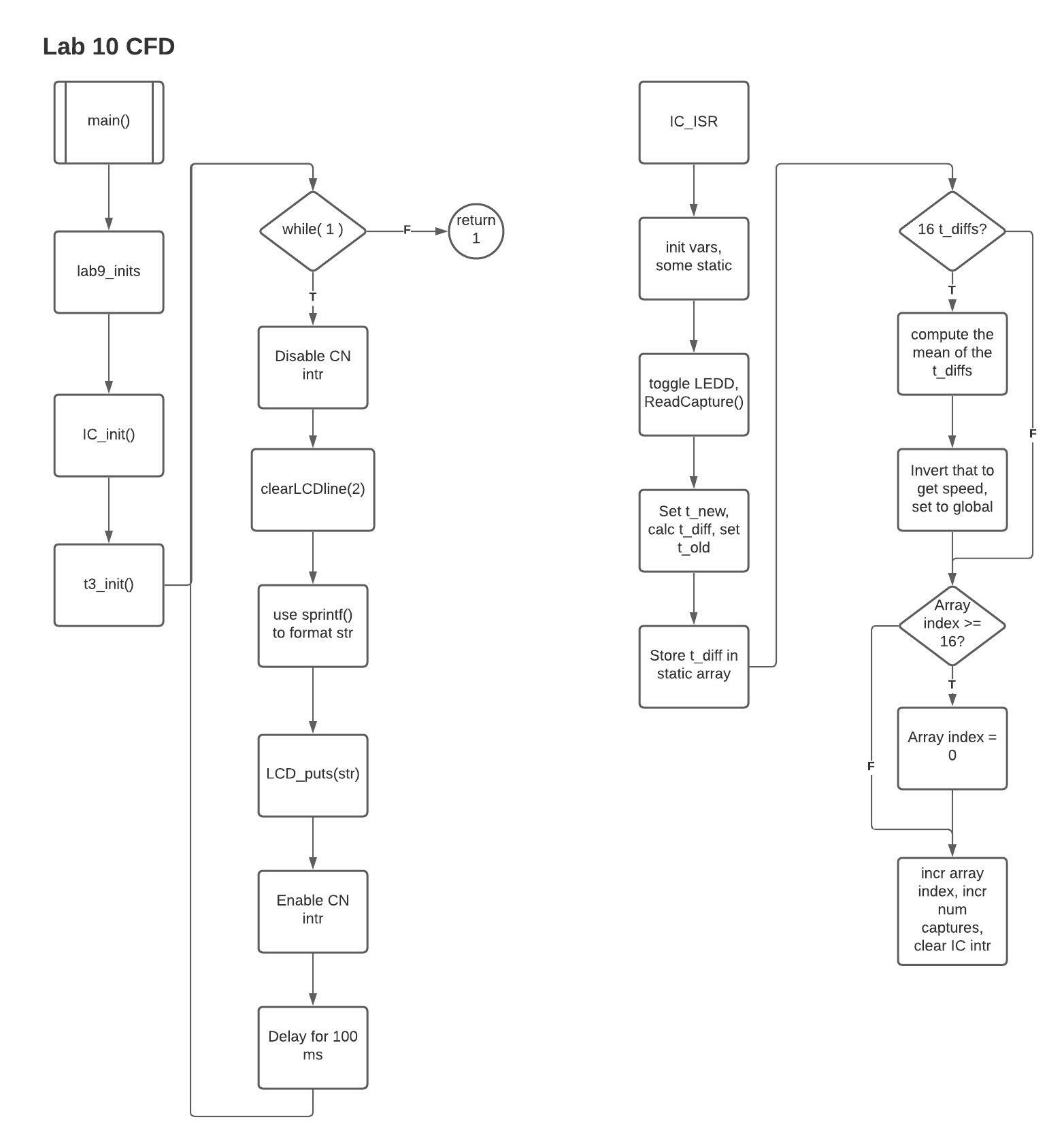
**Listing 4. LCDlib.c and pwm.c:**

The only change made to pwm.c is evident in the data flow diagram. When outputting the duty cycle, we cleared just the first line of the LCD instead of the entire LCD. We needed this so we could output the rps on the 2nd line of the LCD without it getting cleared when outputting the duty cycle, and vice versa as we’ll see shortly.

That discussion brings us to our new LCD functions: ClearLCDline() and ChangeCursorLoc(). As seen in main(), ClearLCDline() takes in a line number (1 or 2) as an argument and clears the specified line, returning the cursor to the line’s beginning after. I used a switch case, as well as a predefined string of 16 white space characters to first change the cursor location to the desired line start, output the empty string, and then reset the cursor.

Looking into the ChangeCursorLoc() function, it also takes a line number as the argument and uses a switch case to either Write the LCD for line 1’s start or line 2’s.

That pretty much covers all functions and file contents implemented to complete lab 10. So, lets preview by looking at the resultant control flow diagram of our efforts.



*• Justify your configuration settings for the input capture peripheral.*

I configured the input capture with the default settings provided in the lab handout. So, I used a 16-bit timer, stopped the input capture during idle debugging, told it to capture on the falling edge, used timer 3 as when to capture, generated an interrupt on each capture, and set its priority to 3.

We'll only use a 16-bit timer, timer 3, since combining two 16-bit timers to get a 32-bit timer would only be appropriate for incredibly widely spaced out events. We stop the input capture during idle debugging, so the LCD doesn’t just start outputting an rps of zero, but rather pauses. I configured it to capture on each falling edge trivially. The rising edge would work just as well, but using both edges would result in greater inaccuracy since the duty cycle is never a perfect 50%. The output capture has a priority of 3 so it’ll take precedence over both of our timers and the change notice interrupt.

*• Derive an expression for calculating motor RPS from the timer count difference t\_DIFF. Where possible, use variables instead of numbers. For example, use f\_PB for the frequency of peripheral bus clock instead of 10 MHz, t\_ps for the timer prescale value instead of 256. State any assumptions and show each step in the derivation. Compare the derived equation with your C code that computes the RPS and comment on any differences.*

Assuming the Timer 3 count is recorded on each transition.

RPS = 1 / second per revolution

T\_DIFF is in units of ticks, so we need to convert it to seconds per revolution

Seconds per revolution = t\_DIFF \* ( t\_ps / f\_PB )

Therefore, RPS = 1 / ( t\_DIFF \* ( t\_ps / f\_PB ) )

Differences between my derived equation and my C code include:

* I have to calculate the average of 16 t\_DIFFs before applying this formula, so t\_DIFF in the formula above is actually the average of 16 t\_DIFFs in my code.
* To prevent incorrect type-casting and resulting truncation, I had to use 1.0 instead of 1 to get a float instead of an integer.
* Along that same vein, I had to typecast/define f\_PB and t\_PS as floats rather than integers.

*• Discuss and defend your moving average implementation. Make sure you include the actual code along with the discussion.*

Code:

*static unsigned int captureNum = 0;*

*static unsigned int index = 0;*

*arrTdiff[ index ] = t\_diff;*

*static const int minCaps = 16;*

*static unsigned short int arrTdiff[ 16 ]; //putting minCaps here didn’t work bc const*

*if( captureNum >= minCaps - 1 )*

*{*

*int sum = 0;*

*int i;*

*for( i = 0; i < minCaps; i++)*

*sum += arrTdiff[ i ];*

*float avg = ((float) sum) / ( (float) minCaps);*

*avg = avg \* (256.0 / 10000000.0); // 256 = t\_ps, 1 M = f\_PB*

*rps = 1.0 / avg;*

*}*

*index++;*

*if( index >= minCaps )*

*index = 0;*

*captureNum++;*

As seen above, I store the 16 t\_diffs in an array before computing their average and resultant rps. This approach is superior to computing each t\_diff’s rps, and then averaging those values because that takes 16 separate calculations to get to rps, while my approach only takes a single one.

I will admit that there are more efficient ways to calculate the moving average, such as when you recalculate the next moving average, only using the difference between the two changed t\_diff values. I prefer my method over this because it introduces more complexe logic. Although this change would minimize the total number of calculations performed, the additional logic makes it more difficult to understand for beginners. In terms of moving average computations, I am a beginner.

My method of resetting the index when it reaches 16 is great because it doesn’t require the use of pointers, but still results in a circular array. A circular array is preferred because it takes up less total memory since we’re overwriting unnecessary values.

**Testing and Validation:**

To demonstrate the TAs that my project worked correctly, I showed that the increase in duty cycle between button combinations also resulted in a higher revolutions per second. One of them pointed out that the graph for speed vs PWM duty cycle in lab handout is fairly close to what rps should be expected, so we compared my LCD output values to that and found them about 100 high. They said they got similar measurements to this when they performed the lab.

3. Connect the logic analyzer probes as follows to demonstrate that the PWM output and Timer 2 interrupts continue to function while the CN interrupt is being served.

a. Channel 0: LEDA – Measures Timer 2 interrupt timing

b. Channel 1: LEDB – Measures the ISR duration that detects a button press

c. Channel 2: LEDC– Measures Timer 3 interrupt timing

d. Channel 3: LEDD – Measures the Input capture timing

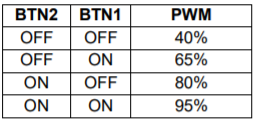
e. Channel 4: PmodHB5\_EN – PWM output signal

f. Channel 5: PmodHB5\_SA – Motor tachometer Phase A

g. Channel 6: PmodHB5\_SB – Motor tachometer Phase B

Capture the logic analyzer screen for the four PWM duty cycle settings specified in Table I.

* already did during last lab, didn't change

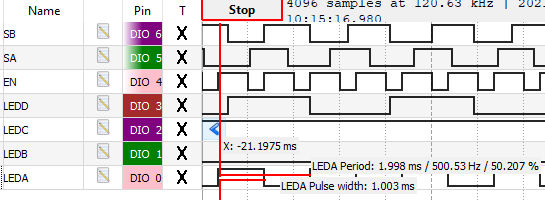


PWM is set to 65%. Measure and record the motor supply voltage (should be 10V ish)

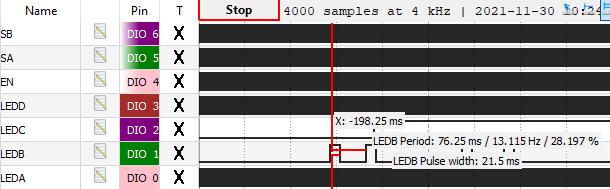
Motor supply Voltage = 9.98V

| Instrumentation | Function | Measurement (Hz) |
| --- | --- | --- |
| LEDA | PWM cycle freq (Timer2) | 500.53 |
| LEDB | Length of CN interrupt | 46.512 |
| LEDC | Input Capture Timer Frequency (Timer3) | 298.06 m |
| LEDD | Input Capture Frequency | 281.84 |
| EN | PWM waveform | 1 k |
| SA | Tachometer Frequency | 546.45 |
| SB | Tachometer Frequency | 546.45 |

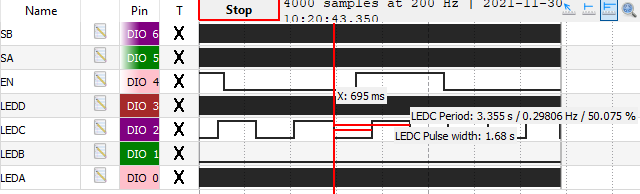
LEDA should toggle at 500 Hz (Timer 2 ISR):



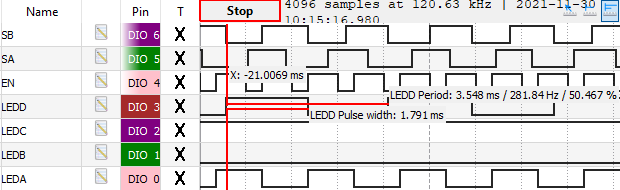
LEDB should toggle around 20 ms (CN ISR debounces for 20ms):



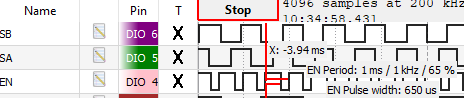
Calculated frequency by: 1 / (21.5 ms / 10^3 ms/s ) = 46.512 Hz

LEDC should have pulse around 1.68 s (Timer 3 ISR):

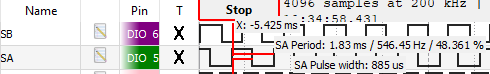
LEDD (input capture ISR):



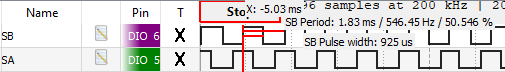
EN (duty cycle of 65):



SA:

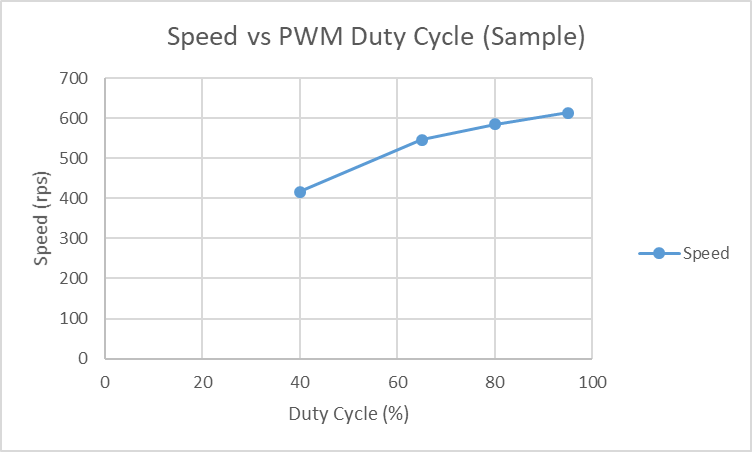


SB:



| BTN2 | BTN1 | PWM duty cycle | SB measurement (Hz) |
| --- | --- | --- | --- |
| OFF | OFF | 40% | 416.67 |
| OFF | ON | 65% | 546.45 |
| ON | OFF | 80% | 584.8 |
| ON | ON | 95% | 613.5 |

Graph the frequency measured at the PmodHB5 SB pin versus the percent PWM duty cycle specified in the table.



| **PWM Duty Cycle (%)** | **V\_M (V)** | **f\_SB (Hz)** | **Motor Speed (RPS)** |
| --- | --- | --- | --- |
| 0 | 10 | 0 | 0 |
| 40 | 9.99 | 416.67 | 415 |
| 65 | 9.98 | 546.45 | 547 |
| 80 | 9.98 | 584.8 | 587 |
| 95 | 9.98 | 613.5 | 612 |

*• What might it imply if your supply voltage drops as your duty cycle increases?*

As seen from the DC power supply’s display, as the duty cycle increases, the supply voltage drops, but the supplied current increases to keep up with the increased transfer rate requested by the higher duty cycle.

*• Plot the motor speed in RPS as a function of PWM duty cycle. Is the relationship linear? Why or why not?*

The RPS vs PWM duty cycle graph shown above does not have a linear relationship because, although not correctly identified in the prelab, the motor has a maximum rps that we’re quickly approaching towards the upper values of the applied duty cycle. Therefore, a logarithmic curve makes much more sense due to this upper bound rps.

**Conclusion:**

In conclusion, we used the PIC32 input capture module to measure the frequency at which the DC motor is turning. All while the DC motor’s speed is changed by a pwm signal as determined by the button combination. We learned about what a moving average is and how to calculate one in several different fashions. This lab was a specific scenario showcasing an instance of how we can use the PIC32’s input capture module to determine the frequency of a signal.

Some limitations of my design include how it’s slightly slower at computing the moving average than an optimal implementation. Due to how we use two buttons, and as a bi-product of lab 9, we can only change the motor speed to four different configurations. We also still can’t change the DC motor’s rotational direction, and we have no way to determine, using the input capture module, which way the DC motor is turning.

*• Implementing feedback:*

*– Describe and define feedback in general.*

For me, feedback resulted in the most differentiation in rps measurements for a PWM duty cycle of 40%. The other three larger PWM duty cycles had around the same variation in rps due to noise. I define feedback in this case as noise generated due to instrumentation and device imperfections. More generally, feedback is an event that occurs when the output of a system is used as input back to the system and isn’t always a bad thing.

*– Describe (using diagrams, pseudo-code, ect.) how you might add feedback to your design to maintain a constant motor speed under a variable mechanical load? More specifically, how might you use the RPS value derived from the Hall effect sensor to adjust the PWM duty cycle such that the motor runs at some specific speed.*

To add feedback, we would have to somehow connect the output back to the input of determination of a characteristic of this mechanical load. We could do this through determining the value calculated for the RPS, calculating an approximate duty cycle for the pwm signal from it, and applying the calculated pwm duty cycle rather than only taking the duty cycle as an input from the buttons. This would surely help pinpoint a constant motor speed.

*• What are the limiting factors when measuring a signal’s frequency with the input capture peripheral as implemented in your project? Starting from the expression derived in the implementation section, derive an expression for the minimum measurable frequency and an expression for maximum measurable frequency. Use the expressions to determine your measurable frequency range.*

The limiting factors when measuring a signals’ frequency using the input capture peripheral are the prescaler of the timer the input capture module uses, and the frequency of the peripheral bus clock, since that’s what’s used to clock the input capture module.

Assuming the timer records on each measurement.

RPS = 1 / ( t\_DIFF \* ( t\_ps / f\_PB ) )

Assuming the minimum and maximum RPS of the DC motor is 525 RPS and 79.5 RPS respectively.

Fsignal = 1 / t\_DIFF

RPS \* (t\_ps / f\_PB) = Fsignal

Assuming the same t\_ps = 256 and f\_PB = 1 MHz as in the project.

Fsignal,min = 79.5 \* ( 256 / 1,000,000 ) = 20.352 mHz

Fsignal,max = 134.4 mHz

Therefore, the measurable frequency range for the DC motor is 20.352 mHz - 134.3 mHz.

*• How might you use the input capture peripheral to build an auto-ranging tachometer? Describe your approach and include an outline of your algorithm.*

Assuming an auto-ranging tachometer is just a tachometer with its range dependent on what it’s measuring, our formula for the Fsignal implies that we essentially already have an auto-ranging tachometer. Therefore, taking measurements as we have in this lab using the input capture peripheral for any device would act as an auto-ranging tachometer.