ECE341

**Lab4 - PIC32 Timers and Multi-rate Scheduling**

Report

**Seth Cram**

**10/05/2021**

**Introduction:**

Goal:

To understand PIC32 timers, their interrupt flag, and its use in how they can be used for multi-rate scheduling.

Background Information:

Timers operate at 10MHz since they are connected to the peripheral bus clock (PBCLK). Since Timer1 is a Type A timer, it’s a 16-bit timer, like the other timers, but with prescale values 1, 8, 64, and 256. A higher prescale value allows us to divide the incoming clock frequency more. This results in the timer only counting 1 tick every ‘x’ clock cycles, with ‘x’ as the prescale value. After turning them on, timers will continue counting until the value loaded into their period register (PRx) is reached. At this point, they’ll set their corresponding timer interrupt flag (TxIF) and start counting again from zero. Care needs to be taken so that the period register is loaded with one less than the number of timer ticks needed, since all timers start counting at zero. Care also needs to be taken to reset the timer flag every time we acknowledge it as set.

Multi-rate processing is when all tasks aren’t executed on the same interval. It’s accomplished by using a timer that counts at a set rate and we execute tasks based on the rate at which its corresponding interrupt flag is triggered. When these interrupt flags are triggered, we decrement counters we use to time events.

*Background information was retrieved from lecture notes, the “Lab4”, and the “Lab3” handouts.*

Plan:

Making use of the code from our previous lab, I’ll modify it to use the PIC32 interrupt flag from Timer1 and counter variables for multi-rate scheduling. First, I’ll need to OpenTimer1(). I placed this in system\_init(), since it’s a part of initializing the system for this project. I made sure to turn the timer on with a prescale value of 1, and with a period register value of 9999. I got the 9999 from the Lab4 handout, where it was calculated using toggles per second, the prescale value, and the peripheral clock frequency. Instead of hardcoding 9999 in, I’ll calculate it in my header file as a constant for readability. Next, we should initialize our two counting variables to zero, so they enter our below discussed conditionals and their values are set from the get-go.

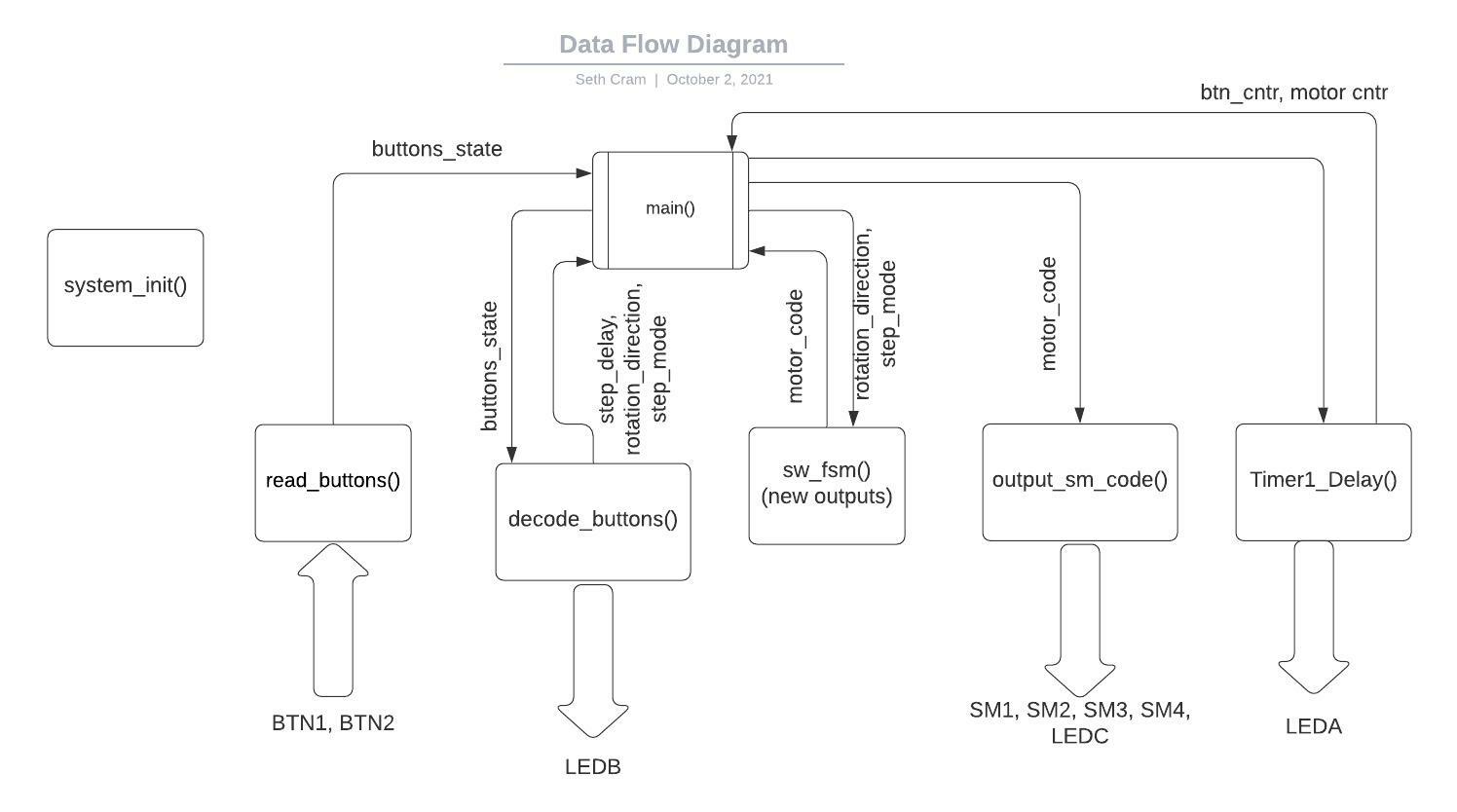
With our system setup, we can enter the while(1) loop. First, we should check if the button counter has reached zero, if so we’ll read and decode the buttons, toggle LEDB, and reset our button counter to 100. Our ‘read\_buttons()’ function is the exact same from the last lab. Within ‘decode\_buttons()’, we’ll have to set the step delay differently. In the lab handout, it specified the different speed of the motor for each button combination. So, we’ll use the formula T\_delay (ms/step) = 60000 (ms/min) / (X rev/min \* 100 steps/rev \* MODE) when MODE is 1 for full stepping and 2 for half stepping, to calculate what we need to set the step delay to in each case. This step delay value is available in main(), since we passed it to decode buttons by reference.

Next in the while(1), if our motor counter is equal to zero, we’ll use our finite state machine and motor output functions the same as we had in the previous lab, toggle LEDC, and reset the motor counter to the step delay found in ‘decode\_buttons()’.

Finally, at the bottom of the while(1) loop, we’ll call ‘Timer1\_delay()’ and pass both our counter variables to it by reference. This function will wait for 1 ms, which is until the Timer1 interrupt flag is triggered, and then clear the interrupt flag, toggle LEDA, and decrement both counter variables. Since we passed in the counter variables by reference, dereferencing them and assigning them a new value will change their value back in main() as well.

This will have the overall effect of reading the buttons once every 100ms, and outputting code to the stepper motor once every 20-40ms depending on the mode and speed of the stepper motor. Therefore, our goal of implementing multi-rate scheduling using Timer1 and its interrupt flag will have been met.

**Implementation Discussion:**

Before implementation, I designed a data flow diagram to get a visual of what functions I’d need to design or modify.

As seen above, I had six functions, five of which were for implementation and the sixth of which was for initialization. Read\_buttons(), sw\_fsm(), and output\_sm\_code() would stay the same from the last lab, but the other three functions would be altered. This is readily apparent with the different inputs and/or outputs from each function.

**Listing 1. Proj4.h:**

Before getting into the affected functions, I’ll identify changes to the header file. This one changes from the previous project’s header file through the addition of several macros used for the setting of the timer 1 period register.

*#define T1\_PRESCALE 1*

*#define TOGGLES\_PER\_SEC 1000*

*#define T1\_TICK (FPB/T1\_PRESCALE/TOGGLES\_PER\_SEC)*

These macros were given to us in the lab 4 handout. With these, also came the addition of the Timer1\_Delay() function prototype, and the subtraction of the sw\_delay() prototype.

*void Timer1\_delay( unsigned int \*btn\_cntr, unsigned int \*motor\_cntr);*

The rest of proj4.h’s contents are the same from lab 3.

**Listing 2. system\_init():**

Moving into main(), as stated in the plan, I added the opening of timer 1 to system\_init().

*OpenTimer1( (T1\_ON | T1\_PS\_1\_1), T1\_TICK-1 ); //turn on timer1 w/ prescale1*

*// load PR1 w/ 9999 bc period of*

*// 1ms reqs it*

*//consts def'd in proj4 header*

Using macros defined in our header file and bitmask macros, I was able to open timer 1 with a prescale value of 1 and a period register calibrated to 1ms.

**Listing 3. New Variables:**

Below system\_init(), but before the while(1) loop, I added variables:

*//timer vars:*

*int btn\_cntr, motor\_cntr;*

*//cnters init'd to zero so reset them properly first iteration:*

*btn\_cntr = 0;*

*motor\_cntr = 0;*

In order to account for execution of button related, and stepper motor related events at different times, we needed counting variables that counted a different amount and reset to different values.

**Listing 4. Newly Structured while(1):**

Now that our setup has finished, it’s time to identify how we implemented multi-scheduling within the while(1) loop. In regards to our buttons:

*if( btn\_cntr <= 0 ) //if time to read btns*

*{*

*btns = read\_buttons();*

*decode\_buttons( btns, &step\_delay, &dir, &mode );*

*LATBINV = LEDB; //toggle LEDB every btn check*

*//reset cntr to check btns every 100ms:*

*btn\_cntr = 100;*

*}*

We first checked if the button counter was equal to or less than zero, which meant it was time to read and decode the buttons. Read\_buttons() and decode\_buttons() are called in the same fasion as in the previous lab. Within this button conditional, we also toggle LEDB for instrumentation purposes. Finally, we reset the button counter to a baseline values of 100, so the buttons are checked once every 100ms.

For the stepper motor, we wanted to check it more often than the buttons to avoid stuttering and since electronics are much more time sensitive than human input.

*if( motor\_cntr <= 0 ) //if time to set motor*

*{*

*sm\_code = sw\_fsm( dir, mode );*

*output\_sm\_code( sm\_code );*

*LATBINV = LEDC; //toggle LEDC every motor write*

*motor\_cntr = step\_delay; //reset cntr*

*}*

Same as before, we only output to the stepper motor when our counter is done being decremented. Both sw\_fsm() and ouput\_sm\_code() are called the same as in lab 3. Then, we toggle LEDC for instrumentation purposes. Finally, we’ll reset the motor counter, but this value it’s reset to is determined earlier within the decode\_buttons() function based on motor speed and step mode. We don’t want to reset to a constant value here because it’d mess up the operation of our stepper motor.

Finally, we call our timer1\_delay() to decrement both counters after 1ms.

*//waits for 1ms + decrements cntrs:*

*Timer1\_delay( &btn\_cntr, &motor\_cntr);*

**Listing 5. decode\_buttons():**

Now, we’ll move onto the internal changes within our while(1) functions. In decode\_buttons(), instead of setting the step\_delay based off of a table, we calculated it as explained in the lab plan. This was done based on the desired motor speed and step mode for each button combination. So, within the button1 case, button2 case, both buttons pressed case, and neither buttons pressed case respectively:

*\*step\_delay = 40; //rpm = 15, FS*

*\*step\_delay = 30; //rpm = 10, HS*

*\*step\_delay = 24; //rpm = 25, FS*

*\*step\_delay = 20; //rpm = 15, HS*

As seen in main, we use this newly calculated step\_delay value to reset our stepper motor counting variable.

**Listing 6. Timer1\_delay():**

This function was a new addition. I takes in the memory locations of both the cutton and stepper motor counting variables to decrement them after waiting 1ms. In order to wait that 1ms, we constantly check if the timer 1 interrupt flag is set yet. If so, we clear the flag to use it for our next 1ms delay, toggle LEDA for instrumentation, and decrement both counting variables.

*//wait for intr flag to be set*

*while( !( INTGetFlag( INT\_T1 ) ));*

*//clear intr flag*

*INTClearFlag( INT\_T1 );*

*LATBINV = LEDA; //toggle LEDA every 1ms*

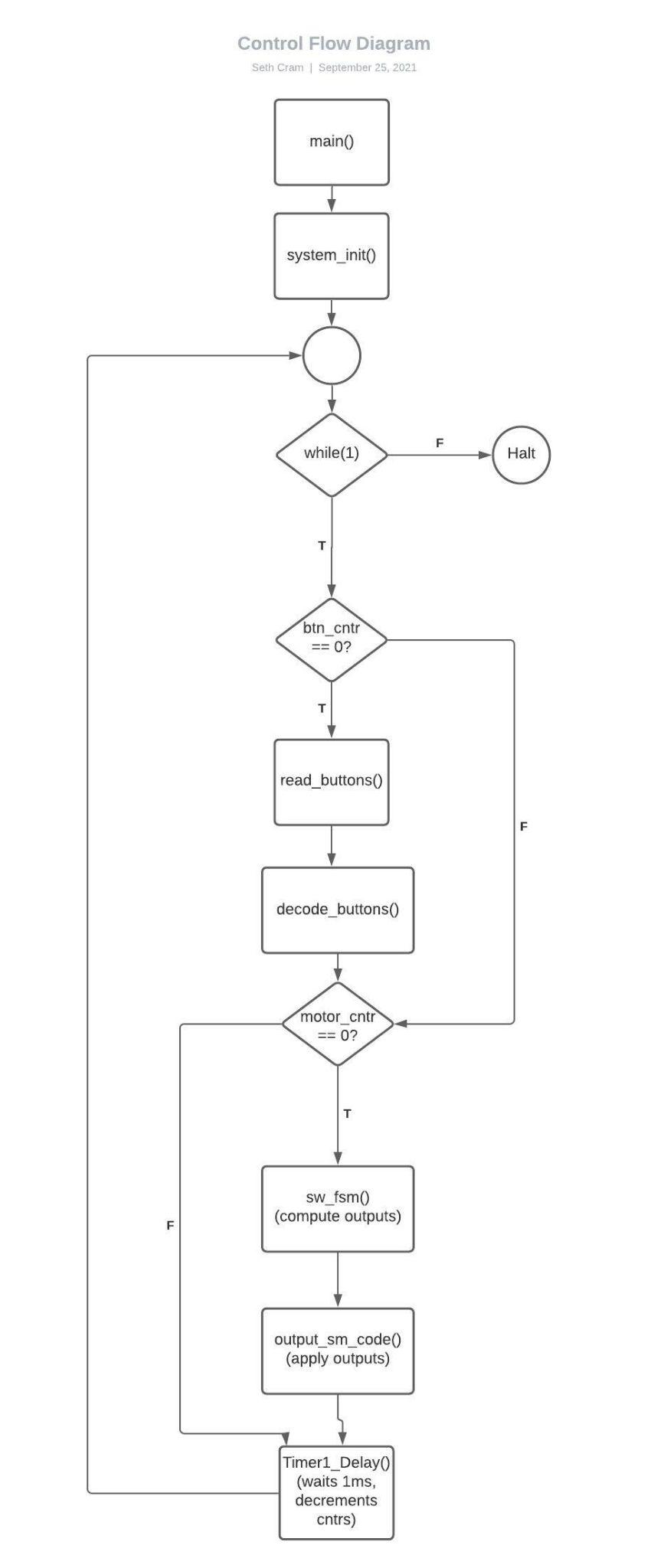
*//decrement cntr vars bc 1ms passed:*

*\*btn\_cntr = \*btn\_cntr - 1;*

*\*motor\_cntr = \*motor\_cntr - 1;*

As seen above, I used the combination of peripheral library functions and the INT\_T1 interrupt source to get and clear the timer1 interrupt flag. In order to decrement the counters, I also need to dereference them since they were passed in through memory addresses. As a result, the values of btn\_cntr and motor\_cntr are also changed in main().

Finally, my control flow diagram models the behavior of the above specified listings:



The sequential execution order is readily apparent, but it's also easy to see how we implemented multi-rate scheduling for this lab in the counter conditionals that jump blocks of code if failed.

*-When discussing the variables used as counters for the button delay and the step delay, justify the type: did you declare the counter variables unsigned int or just int? Justify your decision.*

I declared my counter variables as regular integers. The main reason I used regular signed integers is because if they’re decremented below zero instead of caught at exactly zero for whatever reason, my checks for if they’re equal to or less than zero will catch them on the next loop. Since, in my program I start my counter variables at a number and then decrement them until zero, I never actually want them to reach a negative value, but it’s a good idea to have fail safes’ incase they do. If I had used unsigned integers for my counting variables and counted down, if my zero checks didn’t catch them for whatever reason, they’d be decremented again. Decrementing an unsigned integer that’s at zero loops it to an unsigned integer’s maximum value. This is detrimental since the event tied to this counter will now only happen once the counter is decremented from an incredibly high value to zero.

*– Do your counter variables count up (from zero to the delay value), or do they count down (they are reset to delay value and terminate at zero)? Justify your decision.*

My counter variables count-down to zero from a reset value because, as stated in the previous question, I use integer counter variables with less than or equal to zero comparisons. If I were to count up to a value from zero, I’d have to flip my conditionals and would probably change my variable types to unsigned since they’d never have a reason to be negative.

*– How do you determine the delay period has expired? Do you check with the <= operator, ==, >=, or something else? What might be an advantage of checking with <= (or >=) compared to checking with ==?*

I determined the delay with the <= operator for just incase the counter variable is decremented below zero and it’s missed. The advantages of checking with a <= compared to a == is for error catching, as previously stated.

**Testing and Validation:**

For the demonstration, I showcase how the stepper motor changes directions whenever button 2 is pressed. I also showed how button 1 and 2 being pressed moved the motor faster. Differentiating between no buttons being pressed and just button 2 being pressed was a bit more of a difficult matter. The speed and the direction of the motor didn’t change, just its step mode so I had to bring up the waveform analyzer and show the period of LEDA, which represented the output period to the stepper motor.

I used the formula, T\_delay (ms/step) = 60000 (ms/min) / (X rev/min \* 100 steps/rev \* MODE)

where MODE is 1 for FS and 2 for HS, to calculate the step delay in the below table.

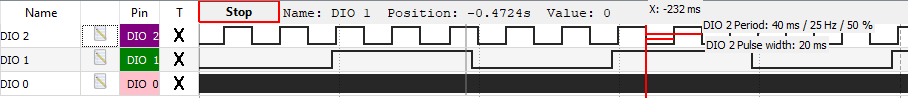
Step Delay Comparison and Calculations:

|  | **Inputs** |  |  | **Controls** |  |
| --- | --- | --- | --- | --- | --- |
| BTN2 | BTN1 | MODE | SPEED (RPM) | Step Delay Calculated (ms) | Step Delay Measured (ms) |
| Off | Off | HS | 15 | 40 | 40 |
| Off | On | FS | 15 | 20 | 20 |
| On | Off | HS | 10 | 30 | 30 |
| On | On | FS | 25 | 24 | 24 |

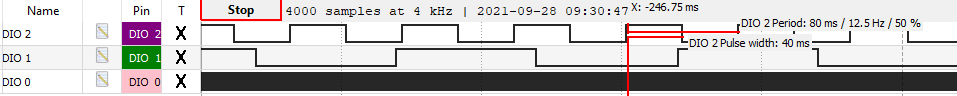
The highlighted red values should be flipped because half step should delay half as much time to make up for its smaller steps, in order to achieve the same rotational speed.

LEDA corresponds to the stepper motor delay, LEDB corresponds to the button checking delay, and LEDC corresponds to the Timer1 decrementing each counting variable every 1ms.

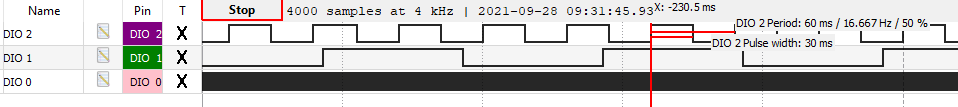
(Step Delay) BTN1 = Off, BTN2 = Off:



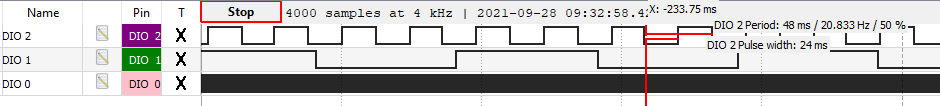
(Step Delay) BTN1 = On, BTN2 = Off:



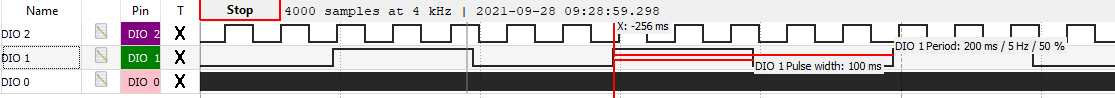
(Step Delay) BTN1 = Off, BTN2 = On:



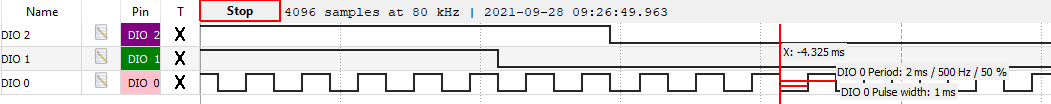
(Step Delay) BTN1 = On, BTN2 = On:



Button Delay:



Timer1 Delay:



**Conclusion:**

From this lab I learned how to implement multi-rate scheduling using a PIC32 timer’s interrupt flag and counting variables. I gained a deeper understanding of the usage of timers within microcontrollers.

Some limitations of our design is that for every single “event” we wish to happen at a different time, we’d need a corresponding counting variable, checking conditional within the while(1) loop, and resetting it within our timer1\_delay() function. This fact makes expanding this method of multi-rate scheduling semi-tedious.

*1. Describe the limitations of using the Timer1 peripheral. What is the longest period that can be measured? What is the shortest period that can be measured? Assume the timer clock source is the peripheral bus clock (FPB) at 10 MHz. Include the derivations for your solutions.*

The longest period that can be measured using Timer1 is 1.68s, found when we set period register 1 to its maximum value of 0xFFFF and Timer1 with its maximum prescale value of 256.

Peripheral Clock: T = 1/f, so 1/(10\*10^6) = 100ns

So, range = 2^16 (ticks/delay) \* 256 (cycles/tick) \* (100 \* 10^-9 (s)) = 1.68 (s/delay)

The shortest period that can be measured using Timer1 is 100ns, found when we set period register 1 to its minimum value of 1, and Timer1 with the lowest prescale value of 1.

Peripheral Clock: T = 100ns

So, resolution = 1 (ticks/delay) \* 1 (cycles/tick) \* (100 \* 10^-9 (s)) = 100 (ns/delay)

*2. How does the period register (PRx) affect the accuracy (resolution) of the timer delay?*

It limits the resolution of the timer delay to a single tick, since you have to set the period register and its minimum value is a whole number of 1. Therefore, as seen in the previous problem, the closest we’re able to get to a desired delay is within 100us, which is incredibly close. (100ns?)

*3. Calculate the change in delay period if the period value written to PR1 changes by one. What is the percent error introduced by this change? Show your work and your derivation.*

Assuming the same setup as used in this lab, PR1 changing from 9999 to 10000 or 9998 would result in a percent error of 0.01% for the desired delay of 1ms. Since, our newly calculated delay period is 1.0001ms. I got these numbers from:

Peripheral Clock: T = 100ns

Delay for PR1 = 10,000 is 10,001 (ticks/delay) \* 1 (cycles/tick) \* (100 \* 10^-9 (s)) = 1.0001ms.

Taking ( (1.0001 \* 10^ -3) - ( 1 \* 10^-3) / ( 1 \* 10^-3) ) \* 100 results in a percent error of 0.01%. Which, in a general equation, is (1 / (PR1 + 1)) \* 100 = % error.

*4. What are the differences between how the core timer and software delay were used, and how the Timer 1 peripheral is used? Specifically, how accurate (consistent) is a sample period that uses the Timer 1 peripheral compared to a sample period that uses the core timer? Hint: think about the time it takes to execute software that is not part of the delay, how (when) the delay period is calculated.*

The Timer1 peripheral is used with an interrupt flag so its delay is much more accurate since it takes into account all the time passed by instructions before the actual function is reached. Meanwhile, a software and hardware assisted delay is affected by how long its previous tasks take, since a delay is only implemented once the function is reached. Therefore, a sample period that uses the Timer 1 peripheral should be much more accurate and closer to the desired value.