

ECE 341 Lab 4

9/10

Zachary DeLuca

February 21st 2023

1 Introduction


scheduling →

In this lab we accomplish the same thing as the previous lab, but using a different coding style this time. Last lab, we controlled a motor using a loop that checked the button output each cycle and wrote to the motor each cycle and delayed the loop based on how fast the motor was meant to spin. This time, the loop will complete once per millisecond and each time it loops it will enter an if clause based on if it is time to either read the buttons or to output to the motor, allowing these tasks to happen at different rates. The buttons do not need to be read at the same rate as the motor is being written to, and the motor needs to be able to have a variant delay to change the speed of the motor.


2 Implementation

For this lab, the sub functions are vary similar to the previous lab, but the main function is the place where of the differences lie. The while loop of the main function is shown below in listing 4.1:

Listing 4.1

```
uint literate = 0; //Stores the value of the buttons
uint shoe = 0; // stores state of the stepper
uint walk = 0; // stores value to output to stepper
uint step_delay = delayed(15); //delay
uint dir = 0; //0 for CCW and 1 for CW
uint mode = 0; // 0 for half and 1 for full
OpenTimer1(T1_ON | T1_PS_1_1, (T1_TICK-1));
uint btnWait = 1000;
uint stepWait = delayed(15);
while(1){
    if(!btnWait){ 
        LATBINV = LEDC;
        literate = readButtons();
        decodeButtons(literate,&step_delay,&dir,&mode);
        btnWait = 100;
    }
    if(!stepWait){
        LATBINV = LEDA;
        shoe = stepperState(dir,mode);
        stepperPush(shoe);
        stepWait = step_delay;
    }
    Timer1_delay(1);
    btnWait--;
    stepWait--;
}
```

In this listing, the core structure of the multi-rate schedule can be seen in each of the if clauses. The button clause is invoked every 100 iterations, which means 100 ms in this case and the stepper output clause is invoked every time a new step needs to be taken. The step delay changes on the speed, which is able to be set to something new every time the button clause is invoked.

The counter variables are unsigned ints so that there are no issues with interpreted negative values as the count should always be a positive number. Making it unsigned also allows for a larger range available to us because there is an additional bit to be used for counting. The variables count down to zero for the reasons that I find the (!variable) syntax concise and appealing and that means we need only set the incremented variable to some new value instead of changing the value of the new if statement benchmark. 

The next listing is the details of the decode buttons function, for the purpose of illustrating the new method of calculating the step delay:

Listing 4.2.1

```

void decodeButtons(int buttons, uint *step_delay,
                  uint *dir, uint *mode){
    int output = 0;
    switch(buttons){
        default:
            *dir = 1;
            *mode = 0;
            *step_delay = delayed(30);
            LATGCLR = LED1 | LED2;
            break;
        case BTN1:
            *dir = 1;
            *mode = 1;
            *step_delay = delayed(15);
            LATGCLR = LED1 | LED2;
            LATGSET=LED1;
            break;
        case BTN2:
            *dir = 0;
            *mode = 0;
            *step_delay = delayed(20);
            LATGCLR = LED1 | LED2;
            LATGSET=LED2;
            break;
        case 0x000000C0:
            *dir = 0;
            *mode = 1;
            *step_delay = delayed(25);
            LATGCLR = LED1 | LED2;
            LATGSET = LED1 | LED2;
            break;
    }
}

```

Listing 4.2.2

```

uint delayed(uint in){
    in = 600/in;
    return in;
}

```

In Listing 4.2.1, the step delay is set using a function that does the math so that the user can simply add in the speed as RPMs instead of having to pre-calculate any values (although the speed is in terms of half-steps, so any speed meant for full steps needs doubling. This could be fixed with an if statement and another passed variable to differentiate between full and half step mode, but as this project does not require a large variety of inputs, we will leave it as such for now)

In Listing 4.2.2, the math of RPM to step delay has been condensed and been allocated to the separate function "delayed". The full math is as follows:

$$Delay(ms) = (\frac{1}{RPM\ input})(\frac{1rev}{100steps})(\frac{60sec}{1min})(\frac{1000ms}{1sec}) = \frac{600}{RPM\ input}$$

3 Testing and Verification

For the testing and the verification, we spun the motor at the various speeds requested and also wrote the same signals out to the LED pins for oscilloscope probing. The timings can be found below in Table 4.1:

Table 4.1:

BTN2	BTN1	Mode	Speed (RPM)	Step Delay (Calculated)	Step Delay (Measured)
Off	OFF	HS	15	40	40
Off	On	FS	15	20	20
On	Off	HS	10	30	29.5
On	On	FS	25	24	24

Below are the four waveform figures showing how each of the above measured values were found. The top waveform in each of the figures is the delay for the stepper motor and the bottom is the delay for the buttons, which does not change between the figures.

Figure 1:

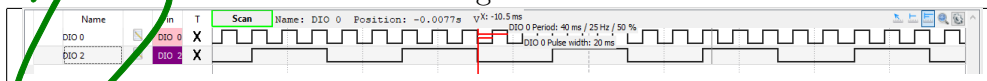


Figure 2:

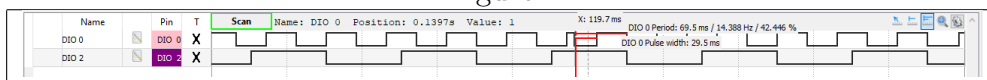


Figure 3:

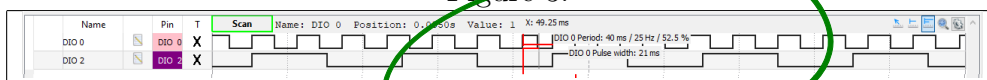
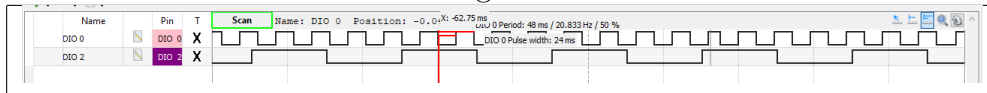


Figure 4:



4 Post Lab Questions

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 Timer1 peripheral is limited in the fact that its counter register is 16 bits wide, meaning it can count fewer ticks than the core timer. At 10MHz, the shortest interval it can detect would be a single tick at that speed meaning:

$$T_{minimum} = \frac{1}{10MHz} = 10^{-7} seconds = 0.0001ms \quad 100ns$$

The maximum is driven by the clock, the size of the counter, and the pre-scaler. So for a timer with a 16 bit wide counter with a 10MHz clock and a 256 pre-scale, the maximum time should be:

$$T_{max} = (\frac{1}{10MHz})(2^{16} ticks)(256) = 1.67 seconds \quad Ahh. \checkmark$$

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

The period register allows for the same delay to happen each time it is called and reset, meaning that the delay should be as or more accurate than the other methods used. If nothing else, the allocation of the timer function to a special register frees up resources making the process probably more accurate.

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.

The change to the period can be shown as a percent difference as the time difference would be dependent on the scale factor, but for the sake of this demonstration, we will assume a scale of 1.

1 ms change (1 off from "delay" value):

$$\frac{1 ms change}{1000ms} = 0.1\%$$

1 tick difference:

$$\frac{1 tick change}{2^{16} ticks} = 0.0015\%$$

$$\frac{1}{10,000} = 0.01\%$$

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 delay was utilized outside of the operations of the buttons and the motor, meaning that it able to clock in less consistently than the previous lab, as each loop took the same time when all the functions were utilized every time. Even though this is the case, the delay is more accurate with the Timer1 delay as it does not need to run the button code or the motor code every iteration.

5 Conclusion

This lab ended successfully with the completion of the multi rate scheduling and the utilization of the Timer1 peripheral. The multi rate schedule appears to be a very useful style for prioritizing functions and sensory readings in a loop with multiple things of variant importance. This technique seems to scale up well to more complex functions for controllers and processors and any kind of tool that utilizes unending loops.