ECE341

**Lab3 - Software Finite State Machine for Stepper Motor**

Report

**Seth Cram**

**09/28/2021**

**Introduction:**

Goal:

Control the speed, rotational direction, and operational mode of a stepper motor using a software finite state machine.

Background Information:

Stepper motors are used to control the angular positioning of a rotor shaft in discrete steps or ticks. The internals of the stepper motor include fixed field windings positioned on the periphery of a rotor with magnets around it. To control the magnitude and direction of the current through these windings, a combination of voltage needs to be applied to the four terminals. This results in 16 possible combinations at the terminals, also known as the possible motor codes. The motor shaft then rotates to a position that minimizes resistance. For this lab, we only use 8 of the 16 possible input combinations.

Current through the coils must be controlled in a specific sequence to move smoothly. The motor moves in either full steps or half step increments, otherwise the transitions aren’t smooth. Tape will be attached to the stepper motor to signify direction in the lab, since we can rotate either clockwise or counter-clockwise. PmodSTEP driver module uses IO PORT B pins 7-10 to control the voltages. Inputs for this lab will be the two buttons as inputs for the motor to configure between clockwise/counter-clockwise and half-step/full-step.

Our software finite state machine will use a switch-case statement and have eight possible cases, one for each present state to calculate its next state. In the main function, we will need to poll the buttons, map them to rotation direction and step mode, find the new motor output, output to the motor, and delay for some time based on the inputs. Our delay function will make sure the stepper motor will be operating at a fixed 15 revolutions per minute.

*Background information is from the lab handouts “lab 3”, “lab 1”, Dr. J’s Powerpoint “proj3-sw\_fsm”, and lecture notes taken in-class.*

Plan:

Making use of the code provided by the lab handout and Dr. J’s powerpoint, we’ll rotate the stepper motor. This will be done depending on the combination of the buttons as input. Before the while loop, I plan on initializing the system and program. I need to set the button bits to inputs, initialize variables, and the Cerebot board.

Before moving on to the while loop, I’ll need to create a separate header file to store the function prototypes and user defined constants for setting options for mode such as ‘HS’ and ‘FS, options for direction such as ‘CW’ and ‘CCW’, and previously found ‘COUNTS\_PER\_MS’ for our software delay function.

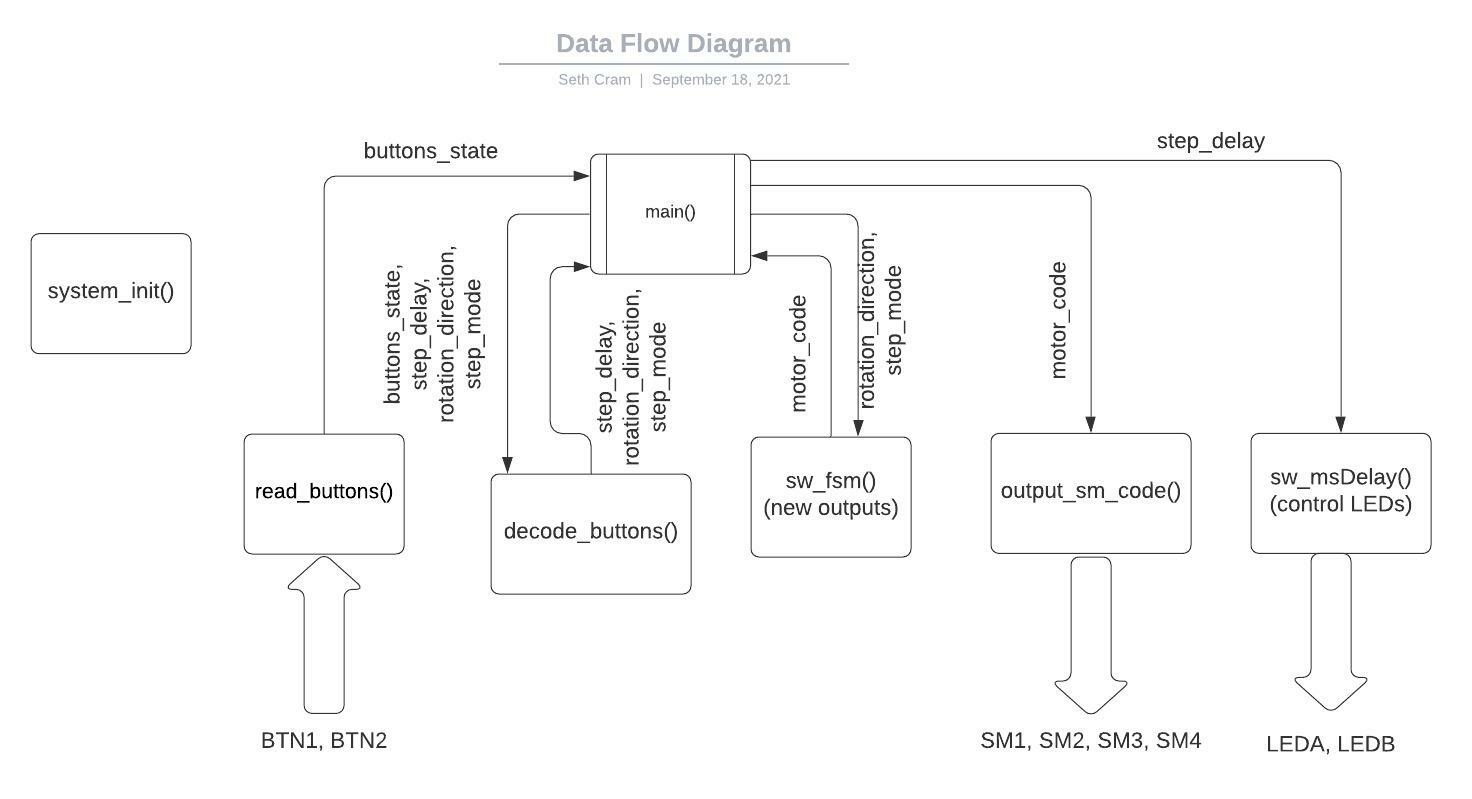
Within the while loop, first I’ll use ‘read\_buttons()’ to store the buttons’ state. Then, I’ll pass ‘decode\_buttons()’ the buttons’ state, and the memory location that the ‘step\_delay’, ‘dir’, and ‘mode’ variables are stored at in memory. Within this function, I’ll receive the memory locations using pointers, and then utilize a switch-case statement with the passed in buttons’ state as its argument. This switch statement will dereference and set the function pointers depending on the buttons’ state. Doing so will change their value back in main().

Next, I’ll return the appropriate stepper motor code from ‘sw\_fsm()’ by passing in the current direction (‘dir’) and step mode (‘mode’) of the motor. Within this state machine function, I’ll use an enumerated type for better state readability, a static variable to keep track of the present state, and a constant to map the present state to a motor code. All of this is done in a switch-case statement that changes the present state based off of the rotational direction and step mode of the motor.

The last new function, ‘output\_sm\_code()’, will receive the stepper motor code output from ‘sw\_fsm()’ and send it to the stepper motor IO pins SM1-SM4. Within this function, we’ll use a Read-Modify-Write operation to keep the state of LEDA and LEDB intact while still setting the stepper motor IO. Our final function, ‘sw\_msDelay() ‘ takes in the variable ‘step\_delay’ that was set in ‘decode\_buttons()’ to ensure a constant 15 revolutions per minute by delaying a certain amount every step/half-step. Here we also toggle LEDA every millisecond and LEDB every delay period for instrumentation. Now at the end of while(), we’ll loop and execute the above described functions and operations within this loop indefinitely.

**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() would stay the same from last lab, but every other implementation function would be altered. This is readily apparent with the different inputs and/or outputs from each function.

**Listing 1. decode\_buttons():**

As stated in the plan, I received the memory locations of several important stepper motor variables using pointers, and then utilized a switch-case statement with the passed in buttons’ state as its argument. Changing my previous if-else statements out for a much cleaner switch-case statement greatly increased the readability of my code, as seen below:

*void decode\_buttons( unsigned int buttons, unsigned int \*step\_delay,*

*unsigned int \*dir, unsigned int \*mode )*

*{*

*switch( buttons )*

*{*

*case BTN1:*

*\*dir = CW;*

*\*mode = HS;*

*\*step\_delay = 20; //HS = 20ms delay*

*break;*

*case BTN2:*

*\*dir = CCW;*

*\*mode = HS;*

*\*step\_delay = 20;*

*break;*

*case BTN1 | BTN2: //both btns pressed*

*\*dir = CCW;*

*\*mode = FS;*

*\*step\_delay = 40; //FS = 40ms delay*

*break;*

*default: //neither btn pressed, or more than 2 pressed*

*\*dir = CW;*

*\*mode = FS;*

*\*step\_delay = 40;*

*break;*

*}*

*}*

Then, I dereferenced the pointers which pointed to the passed in memory locations in order to change their stored values here and back in main(). This was done in order to configure the stepper motor based off of the button(s) being currently pressed. This implementation was a little rocky. As soon as I got into the lab, there was immediately a problem with my code: the constants ‘CW’, ‘CCW’, ‘HS’, and ‘FS’ weren’t already initialized in any of the header files!

**Listing 2. Proj3.h:**

So, I went ahead and constructed a header file of my own, defining the below constants:

*#define HS 1*

*#define FS 2*

*#define CW 1*

*#define CCW 2*

Since the variables these values were stored in were never compared to anything other than the four constants I have listed above, the actual value I assigned to them was trivial.

Within this header file I also made sure to include prototypes of all my functions and defined ‘COUNTS\_PER\_MS’ as found in the previous lab.

*#define COUNTS\_PER\_MS 8889*

*void system\_init (void); /\* hardware initialization \*/*

*int read\_buttons(void);*

*void decode\_buttons( unsigned int buttons, unsigned int \*step\_delay,*

*unsigned int \*dir, unsigned int \*mode );*

*unsigned int sw\_fsm( unsigned int dir, unsigned int mode );*

*void output\_sm\_code( unsigned int sm\_code );*

*void sw\_msDelay(unsigned int mS);*

**Listing 3. sw\_fsm():**

Most of this code was retrieved from Dr. J’s powerpoint. It takes in the direction of stepper motor and its mode as unsigned integers:

*unsigned int sw\_fsm( unsigned int dir, unsigned int mode )*

It then goes on to defined state variables for readability, and a static present state variable:

*enum { S0 = 0, S0\_5, S1, S1\_5, S2, S2\_5, S3, S3\_5 }; /\* Declaration of states \*/*

*static unsigned int presState;*

It utilized a static variable so that every time we enter the function we remember what the present state is. We are also required to use the table presented in the lab report to specify the values taken on by ‘sm\_code’ at each of its 0-7 index places. We needed this in order to return a valid stepper motor code for the present state:

*const unsigned int sm\_code[] = { 0x02, 0x0A, 0x08, 0x09, 0x01, 0x05, 0x04, 0x06 };*

Finally getting around to the actual determination of state, I used a switch-case statement to create my finite state machine. Within each present state case, I had to check the current direction, and based off of that, check the current stepping mode. Based on both of these things, I would change the present state accordingly. For example, the first case statement contained:

*if ( dir == CW ) //if rotting CW*

*{*

*if( mode == HS )*

*{*

*presState = S0\_5;*

*}*

*else //full stepping*

*{*

*presState = S1;*

*}*

*}*

*else //CCW*

*{*

*if( mode == HS )*

*{*

*presState = S3\_5;*

*}*

*else //full stepping*

*{*

*presState = S3;*

*}*

*}*

Then, at the end of the function, we’d return a valid stepper motor code:

*return sm\_code[ presState ]; /\* Return next as curr state \*/*

This is where we utilized the above defined ‘sm\_code’ indices to properly return the correct stepper motor code. The implementation of this function was rather straightforward after viewing the lab handout table regarding which stepper motor code corresponds to what mode and direction.

**Listing 4. output\_sm\_code():**

As specified during the plan, this function needed to utilize a Read-Write-Modify in order to output the stepper motor code to the motor without affecting LEDA and LEDB, since they and SM1-4 were all in Port B. The below code is an altered version of the operation presented in a Lab 1 Appendix:

int temp, new\_data, mask;

mask = SM\_COILS; //SM\_COILS = (SM1 | SM2 | SM3 | SM4)

new\_data = (sm\_code << 7); //our new data is the step motor code;

// we shift it 7 bits to left since

// motor code is pins 7-10

// bc sm\_code = 1st 4 bits

temp = LATB; //read all 16 port B bits

temp = temp & ~mask; //clear all bits that need to be set by our stepper motor code

new\_data = new\_data & mask; //clear all non SM1-4 bits

temp = temp | new\_data; //recombine curr bits with new bits

LATB = temp;

Our ‘mask’ is specified as the bits we want to change within the Port. For this, we use a new constant called ‘SM\_COILS’ which is just the bitmask of all the stepper motor output bits. Then, we shift our passed in ‘sm\_code’ seven bits to the left, since we want to change bits 7-10 of PortB. At this point, we need to read from LATB, clear the bits we need to set in both the variables LATB is stored in and our read-in stepper motor code, just to make sure we’re writing correctly. Combining our shifted stepper motor code with the read-in LATB, we can then safely write it back to LATB without changing the state of LEDA or LEDB.

The Read-Write-Modify operation was conceptually challenging for me to grasp. It took some time of in-class discussion and searching in my free time to understand that when we write to a port, all the bits we don’t specify as one get cleared. After this revelation, the implementation of this function was much easier.

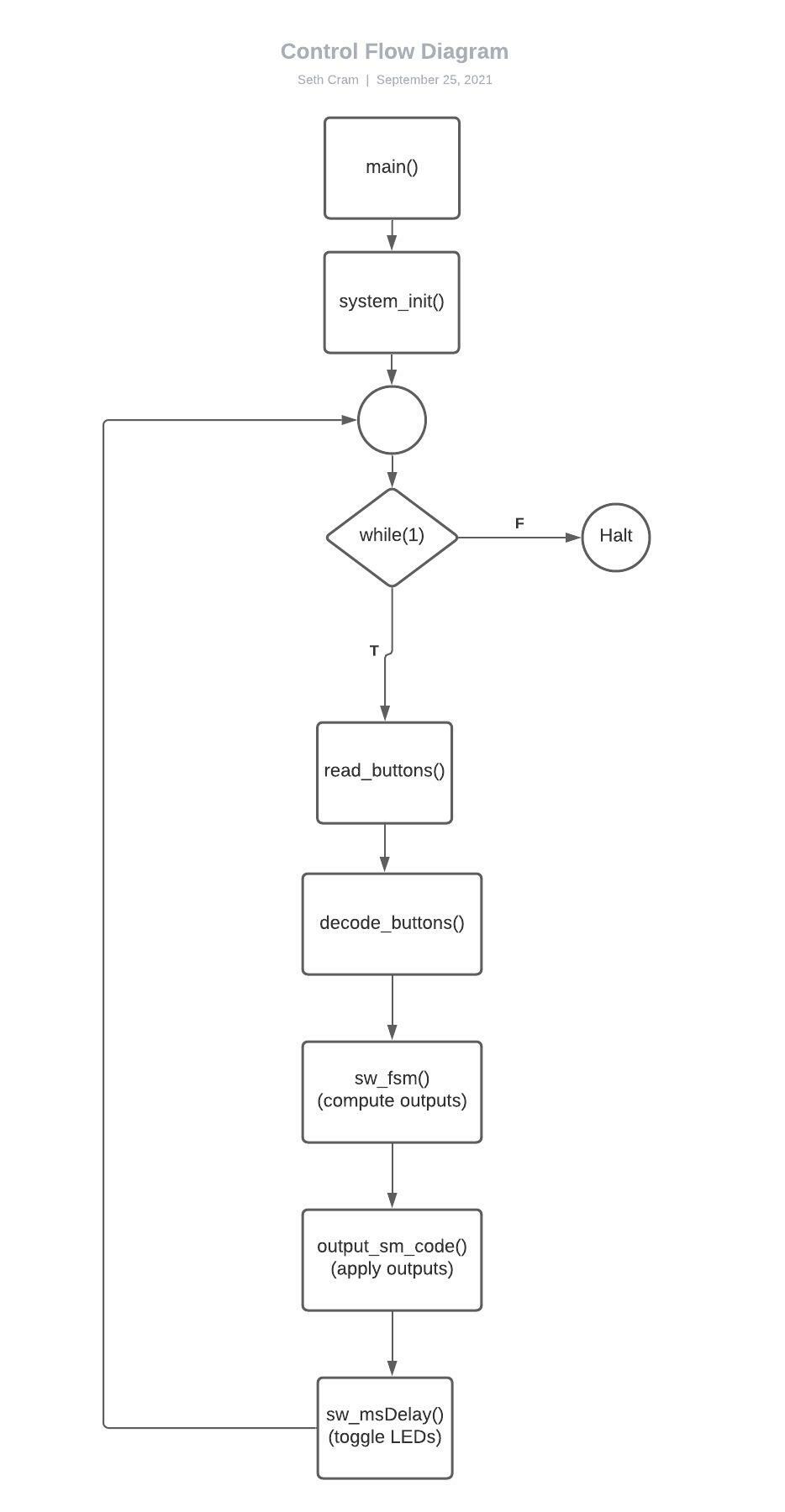
**Listing 5. sw\_msDelay():**

The only change from last lab was toggling LEDB at the end of the software delay in order to flip this light every delay period:

LATBINV = LEDB; /\* Toggle LEDB each delay period \*/

Before, I had done this in main(), but felt it fit better within the delay function.

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



The sequential execution order is readily apparent. From the prelab, I change the diagram to better incorporate the decision made by the while(1) loop. This was done after viewing more control flow diagram examples.

**Testing and Validation:**

For the demonstration to the TA, I showcase how the stepper motor changes directions from clockwise to counter-clockwise when pressing button 2, or button 1 and 2. Releasing button 2 resulted in the motor correctly going back to clockwise rotation.

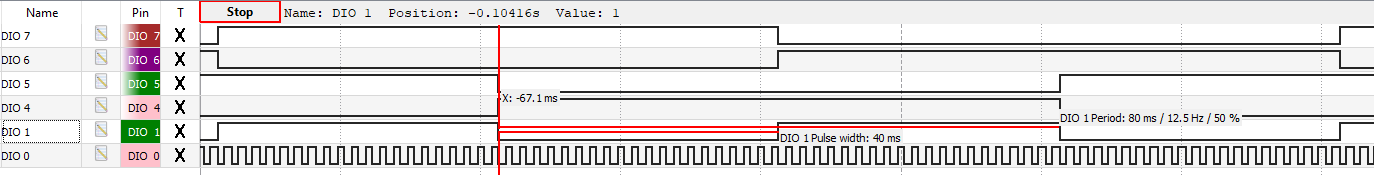
Showcasing how the stepping mode changed was a bit more difficult. There was no discernable change displayed by the stepper motor between the step modes since we ran it at a constant 15 RPMs. So, I brought out the waveform analyzer application and showed how when only button 1 or only button 2 was being pressed, the period of the LEDB waveform was 20ms rather than its normal, full-stepping 40ms. LEDB’s period corresponded to the step delay, so this method worked perfectly.

Measured Step Delay Table:

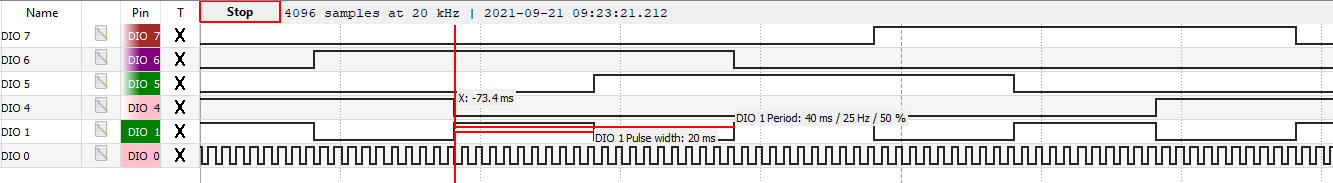
|  | **Inputs** |  | **Control Modes** |  |
| --- | --- | --- | --- | --- |
| BTN2 | BTN1 | Step Mode | Step Delay  Calculated | Step Delay  Measured |
| Off | Off | FS | 40ms | 40ms occasionally 40.25ms |
| Off | On | HS | 20ms | 20ms occasionally 20.02ms |
| On | Off | HS | 20ms | 20ms occasionally 20.25ms |
| On | On | FS | 40ms | 40ms |

Measured Step Delay Table’s Corresponding Tests:

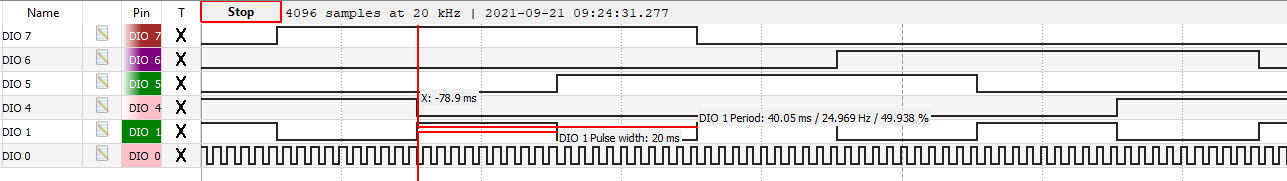
BTN2 off, BTN3 off:



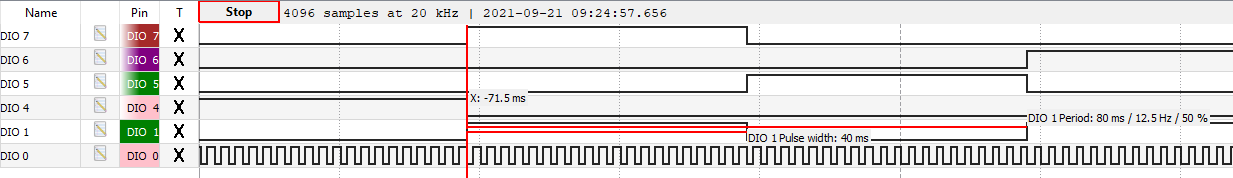
BTN2 off, BTN1 on:



BTN2 on, BTN1 off:



BTN2 on, BTN1 on:

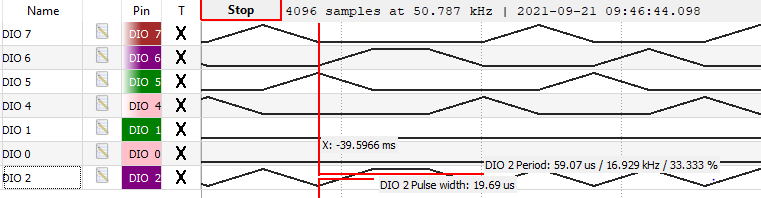


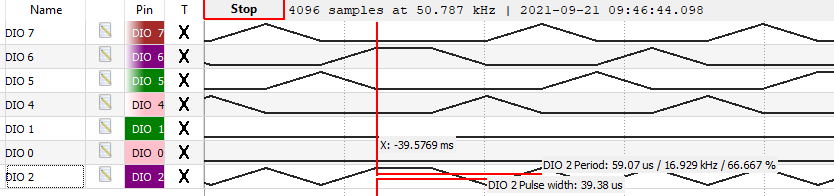
For the above tests:

*DIO4-7 = LEDE-LEDH (SM1-SM4)*

*DIO0-1 = LEDA-LEDB*

To measure the execution time for code that isn’t part of the delay period, I implemented LEDC to be flipped at the end of each while loop and commented out the delay function. Therefore, a single period of LEDC would be two loops of code since it’d turn on at the end of one loop execution and turn off at the end of the next loop execution. So, to measure the execution time of the code outside of the delay function a single time, I’d have to divide the period of LEDC by two.





As seen above, the period varied from 19.69us to 39.39us. So, on average, the time to execute two loops of code is 29.52us. In conclusion, the time to execute code that is not part of the delay is 14.77us.

**Conclusion:**

From this lab I learned how to control the speed, rotational direction, and operational mode of a stepper motor using a software finite state machine. I gained a deeper understanding of how a Read-Write-Modify operation is useful for preserving the current state of bits when writing to a port. I now understand how finite state machines can be used in more than just FPGAs.

Some of the limitations our design has is that the Read-Write-Modify operation is susceptible to outside influence changing the value of port bits after we read them. So, we have to be careful dealing with PortB bits. As seen in the Testing and Validation section, our delay also isn’t exact since the execution of the non-delay code is 14.77us. Now that we’ve discussed the constraints and results of the lab, it’s time to answer the end of lab questions.

*What are some various methods for implementing a FSM in C? When might one method be better than another in a given situation? Describe the advantages and disadvantages of each method. What method do you like best? You will likely have to do some research, make sure to cite your sources.*

Methods to implement a FSM in C include using a switch-case statement like we did in this lab, and using a 2D-array containing function pointers and state transition lookup rules. I was unable to find any more different implementations than these two for a FSM in C. Using function pointers seems more advantageous for large FSMs since they require less lines of code. The disadvantage of using function pointers is the steeper learning curve since they require knowledge of both the internal workings of pointers and structs. A switch-case statement is more advantageous when considering a small number of total states, and it is better suited for beginners because of the use of basic conditionals such as if-else and the switch-case statement itself. The switch-case statement method is disadvantageous because of its bulk.

I recovered information regarding the function pointer method from <https://stackoverflow.com/questions/1371460/state-machines-tutorials>.

*What are the biggest differences between an FSM implemented on a microcontroller and an FSM implemented on an FPGA?*

A FSM implemented on a microcontroller is done solely through software, but a FSM implemented on an FPGA needs to incorporate hardware aspects into its workings. This is the case for the FPGA because without a fixed hardware structure like a microcontroller has, an FPGA requires the user to construct hardware to support the FSM. The flexibility provided by the FPGA means the microcontroller is better for specific functions such as this.

*How often are the button inputs sampled? Discuss the consequences of this and briefly describe one possible way of solving the problem*

The delay routine waits for 20 or 40ms depending on the step delay, which is determined by the step mode. On top of this, the code executed outside of the delay routine takes at maximum 19.695us. So, the button inputs are sampled, in the worst case, about every 20.020ms when the stepper motor is in half step mode. When the stepper motor is in full-step mode, the buttons are sampled, in the worst case, around every 40.020ms.

The consequences of this are the persistence of the buttons being pressed “event” has to last for at least 20.020ms when the stepper motor is in half step mode, or 40.020ms when the stepper motor is in half step mode. So, the buttons have this much of a maximum latency to reach the program. Otherwise, the button inputs are completely missed by the polling program and have to either last until the next button sampling, or risk not being read at all.

One possible way to solve this problem is through using interrupts to drag the processor’s attention away from running our instructions to read the buttons. This would result in the program not having to manually check the state of the buttons every loop cycle, and overall more efficient execution and far better worst case latency of reading the buttons.