Lab 3: Software Based Finite State Machines

Report

**Owen Blair  
9/28/2021**

Contents

[Introduction 2](#_Toc83668835)

[Implementation 2](#_Toc83668836)

[Testing and Verification 2](#_Toc83668837)

[Questions 2](#_Toc83668838)

[Conclusion 2](#_Toc83668839)

# Introduction

Lab 3 Prelab: Software Based Finite State Machines

Owen Blair

# **Goal and Background Information**

The goal for Lab 3 is to move a stepper motor at a constant 15 RPM. The mode and direction that the stepper motor moves in is determined by the inputs from button 1 and button 2.

New subsystems and concepts used are Enums, software finite state machines, and open loop control. Enums is a defined set of values that a specific Enum variable can be. For example, if you have written enum apartment\_numbers{bottom\_floor = 1, middle\_floor = 2, top\_floor = 3}  then you could use them to ensure that only certain values were used when using the Enum when executing instructions.  Software finite state machines are like FSM that were defined in digital logic (ECE-240) but are defined using software and switch statements instead of logic gates. Lastly, open loop control or real-time open loop control implies that this type of scheduling is time dependent, and tasks must happen at specific times. For lab 3, the code will read, decode, implement the software FSM, and write to the stepper motor as fast as possible but then will delay using the sw\_delay function. This kind of timing is real-time loop control.

For most of the lab I intend to copy  and paste the functions needed because I have already written most of them. This includes functions like system\_init(), sw\_delay(), and read\_buttons(). The function decode\_buttons() will be mostly identical with a little modification to output the needed delay to maintain the 15 RPM depending on the step mode. The functions stepper\_state\_machine() and output\_to\_stepper\_motor() will need to be written from scratch.

The stepper\_state\_machine() will take two inputs, the current state of the stepper motor and the decoded inputs from the buttons and will output a next state. The logic of the function will be made of a switch statement that is controlled by the current state and then if statements that output the next state depending on the button inputs.

Output\_to\_stepper\_motor() will utilize an Enum that defines the different instructions that can be written to the stepper motor. This way anything written to the stepper motor using the Enum can’t write an invalid instruction to the stepper motor control board.

**Diagram

Description automatically generatedData Flow Diagram**

**Diagram

Description automatically generatedControl Flow Diagram**

# Implementation

The implementation of the code for lab 3 did not go as planned. The extra time provided by the TAs on Thursday was essential. There were several bugs that inhibited the way my program ran. The first one I came across was that I had tried to assign an unsigned integer to zero using an enumeration. The second was a semicolon that I was missing in configbits.h that I accidently deleted when looking for bitmasks.

The new code for this lab consisted of three functions. A modified decodeButtons() function from the previous lab, a finite state machine to control the stepper motor’s behavior called stepperStateMachine(), and an outputToStepper() function to write the new state to the stepper motor. The other function used for this lab is sw\_delay() and is a software delay but is described in a previous lab report.

New inputs and outputs were added to the decodeButtons() function for this lab. All the new inputs are pointers to variables that were returned separately. The reason for this is that the decodeButtons() function needed to return several new discrete variables to control the stepper motor. This is things like what direction was specified and what the specified movement mode is for the motor from the button input. Using pointer allowed me to change the variables inside the function and still have the changes be remembered when the CPU exited the function. The structure of the decodeButtons() function is a switch statement of each button combination and each case would set the variable outputs to what the user specified through the buttons. The outputs of the functions differ between lab 3’s decodeButtons() function and lab 2’s because lab 2’s decode buttons returns an integer while lab 3’s function returns void. The reason that the decodeButtons() function in lab 3 returns void is because the variables are modified using pointers and it was unnecessary to return a value that was modified using pointers. The code for this can be seen in Listing 1.

The code for the finite state machine was surprisingly long and seemed like it could be more efficient about how the if statements were implemented. The way that the newly implemented finite state machine, called stepperStateMachine(), operated was by having inputs of the direction (the desired direction from the user and decoded from decodeButtons() function), &stepState (the current stepper motor state variable’s pointer), and the mode (the desired movement mode from the user and decided from the decodeButtons() function. This can be seen in Listing 2.

Lastly, an outputToStepper() function was made to write the desired instructions to the stepper motor. The outputToStepper() function took one argument, stepState, and returned void. The stepState variable that the function received was the value returned from the stepperStateMachine() function. To make the code easy to read and debug, the different ‘states’ the stepper motor could be in were implemented using an enumeration (specified in the introduction). This meant that all the function needed to do was putput the value of the stepperState to the IO pins the stepper motor control board were attached to. The instructions sent to the stepper motor control board used a read, modify, write style of instruction. This can be seen in Listing 3.

# Testing and Verification

Two forms of verification were used to test and verify that the code written for this lab was working correctly. The first is visual inspection. This consisted of pushing the buttons on the MX7 board and seeing if the stepper motor acted accordingly. The behavior was checked using Table 2 from the lab handout. The speed at which the stepper motor was also timed using a wristwatch to see how long it took to turn a quarter of a circle. If the time it took to turn a quarter of a circle, then the RPM of the stepper motor was about 15.

The RPM of the stepper motor was also confirmed by the second form of testing and program verification. The second method used the oscilloscope on the Digilent explorer and the waveform program to examine the signals asserted to LED A and LED B on the stepper motor control board. LED A was toggled every time the sw\_delay() function went through a one millisecond wait. The results of validation and testing of the sw\_delay() function can be seen in a previous lab. LED B was toggled every time an instruction was sent to the stepper motor. If the program was working correctly, LED B would be toggled every 20 milliseconds if moving in half step mode and 40 milliseconds if moving in full step mode. When tested, the delay between toggles of the LED B state in half step mode was 20.02 milliseconds and the delay between toggles for full step mode was 40 milliseconds. The half step waveform can be seen in Figure 1 and the full step waveform can be seen in Figure 2. The recorded waveform times can also be seen in Table 1 below.

## Table 1

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Inputs | | Stepper Parameters | | |
| BTN 1 | BTN 2 | Step Mode | Desired Step Delay | Measured Step Delay |
| Off | Off | Full Step | 40 ms | 40 ms |
| On | Off | Half Step | 20 ms | 20.02 ms |
| Off | On | Half Step | 20 ms | 20.02 ms |
| On | On | Full Step | 40 ms | 40 ms |

## Graphical user interface, application, Word Description automatically generatedFigure 1

## Graphical user interface Description automatically generated with low confidenceFigure 2

# Conclusion

Two methods of implementing a finite state machine in C include nested conditional statements, like switch or if statements, or using arrays of structures[[1]](#footnote-1). The best time to use nested switch/if statements is when you need something that is easily readable and quick to set up. Using conditional statements are not good to use when there are many states within a finite state machine because the length of the code is directly proportional to the number of states. Arrays of structures are good to use when there are many different states to the finite state machine because all the different values within a state can be defined within the definition of each array structure entry. With lots of different states this can make code easy to read and save on lines of code written. This is done by the finite state machine returning a structure with the state information contained inside and then using objected oriented programming to assign each output to the value in the corresponding state structure.

The difference between implementing a finite state machine on an FPGA verses a microcontroller comes down to the cast that a microcontroller has limited processing power where the FPGA has limited space. There are also differences in how finite state machine code is executed between the two. A FPGA can process multiple commands at the same time because it relies on programed logic gates where a microcontroller can only execute one instruction at a time[[2]](#footnote-2). Another difference in implementation is the code in which the finite state machine is written in. For FPGAs the code is written in VHDL or Verilog where the microcontroller is programmed in C or C++.

In the code for the program, the button inputs are sampled every 20 or 40 milliseconds. The consequences of this is that the stepper motor will respond faster to button inputs after a half step instruction then it will after a full step instruction. This is because a half step instruction used the sw\_delay() function to wait for 20 milliseconds where the full step instruction makes the CPU wait 40 milliseconds before continuing. An implication of this is that the CPU is more likely to miss a button change event during a full step move iteration than a half step move iteration.

# Listings

## Listing 1

/\* decode\_buttons Function Description \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\* NOTES: Refer to Cerebot MX7cK data sheet for button bit positions

\* END DESCRIPTION \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

void decode\_buttons(unsigned int buttons, unsigned int \*delay, unsigned int \*mode, unsigned int \*direction)

{

switch(buttons)

{

case 0://All off

\*direction = CW;

\*mode = FS;

\*delay = 40;

break;

case 1://BTN1 on, BTN2 off

\*direction = CW;

\*mode = HS;

\*delay = 20;

break;

case 2://BTN1 off, BTN2 on

\*direction = CCW;

\*mode = HS;

\*delay = 20;

break;

case 3://BTN1 on, BTN on

\*direction = CCW;

\*mode = FS;

\*delay = 40;

break;

}

}

## Listing 2

/\*

stepperStatemachine function description:

\* This function takes the current state of the stepper and the direction

\* that the stepper motor needs to move in.

\*/

unsigned int stepperStateMachine(unsigned int direction, unsigned int stepState, unsigned int mode)

{

switch(stepState)

{

case s0\_5:

if (direction == CW)

{

if(mode == FS)

{

stepState = s1\_5;

}

else

{

stepState = s1;

}

}

else //CCW

{

if(mode == FS)

{

stepState = s3\_5;

}

else

{

stepState = s4;

}

}

break;

case s1:

if (direction == CW)

{

if(mode == FS)

{

stepState = s2;

}

else

{

stepState = s1\_5;

}

}

else //CCW

{

if(mode = FS)

{

stepState = s4;

}

else

{

stepState = s0\_5;

}

}

break;

case s1\_5:

if (direction == CW)

{

if(mode ==FS)

{

stepState = s2\_5;

}

else

{

stepState = s2;

}

}

else //CCW

{

if(mode == FS)

{

stepState = s0\_5;

}

else

{

stepState = s1;

}

}

break;

case s2:

if (direction == CW)

{

if(mode == FS)

{

stepState = s3;

}

else

{

stepState = s2\_5;

}

}

else //CCW

{

if(mode == FS)

{

stepState = s1;

}

else

{

stepState = s1\_5;

}

}

break;

case s2\_5:

if (direction == CW)

{

if(mode == FS)

{

stepState = s3\_5;

}

else

{

stepState = s3;

}

}

else //CCW

{

if(mode == FS)

{

stepState = s1\_5;

}

else

{

stepState = s2;

}

}

break;

case s3:

if (direction == CW)

{

if(mode == FS)

{

stepState = s4;

}

else

{

stepState = s3\_5;

}

}

else //CCW

{

if(mode == FS)

{

stepState = s2;

}

else

{

stepState = s2\_5;

}

}

break;

case s3\_5:

if (direction == CW)

{

if(mode == FS)

{

stepState = s0\_5;

}

else

{

stepState = s4;

}

}

else //CCW

{

if(mode == FS)

{

stepState = s2\_5;

}

else

{

stepState = s3;

}

}

break;

case s4:

if (direction == CW)

{

if(mode == FS)

{

stepState = s1;

}

else

{

stepState = s0\_5;

}

}

else //CCW

{

if(mode == FS)

{

stepState = s3;

}

else

{

stepState = s3\_5;

}

}

break;

return stepState;

}

}

## Listing 3

/\*

This function takes a state of the stepper motor and moves it!

\*/

void outputToStepper(unsigned int stepState)

{

unsigned int x = stepState;

x = PORTB;

x = x & ~SM\_COILS;

x = x | (stepState<<7);

LATB = x;

LATBINV = LEDB; // Toggle LEDB for each new stepper movement and used

//for instrumentation

}

1. <https://aticleworld.com/state-machine-using-c/> [↑](#footnote-ref-1)
2. <https://www.electronicslovers.com/2018/12/microprocessor-fpga-vs-microcontrollers.html> [↑](#footnote-ref-2)