8/10

ECE 341 Lab 3

Zachary DeLuca

February 14th 2023

might want to clarify sw-based

---

1 Introduction

Hmm.

In this lab we will use the controller to control a stepper motor. The most interesting part of this lab is that the stepper motor exists and moves in discrete steps, meaning it can be controlled using a finite state machine. This lab institutes a single rate scheduled while loop that simply uses the read-modify-write structure.

so

As opposed to doesn't exist?

2 Implementation

To implement the functions of the lab, the main loop was set up in the way shown in listing 3.1 below.

In the header file, #define uint unsigned int was used to reduce the amount of times unsigned int was needed to be written.

Hmm

you should work on improving your technical communication.

Listing 3.1

```
int main() {
        system_init (); /* Setup system Hardware*/
        uint literate = 0; //Stores the value of the buttons  because it's well read
        uint shoe = 0; // stores state of the stepper
        uint walk = 0; // stores value to output to stepper
        uint step_delay = 0; //delay
        uint dir = 0;
        uint mode = 0;
                        while(1) {
                literate = readButtons();
                decodeButtons(literate &step_delay &dir &mode);
                shoe = stepperState(dir mode);
                stepperPush(shoe);
                delay(cdelay);
                                }
        return 0;
/* Returning a value is expected but this statement      * should never execute */
}
```

The first listing just shows the general structure of the lab, and the sub functions show more clearly the meat of what is actually happening. Least interestingly is the read buttons function listed in Listing 3.2 below:

Listing 3.2

```
int readButtons() {
        int mask = (BIT_6 — BIT_7);
        int garbo = PORTG & mask;
        return garbo;
        }
```

Following the buttons being read, the next step is the buttons being decoded which is shown in the next listing:

Listing 3.3

```
void decodeButtons(int buttons  uint *step_delay uint * dir uint * mode){
    int output = 0;
    switch(buttons) {
        case BTN1:
        *dir = 1;
        *mode = 0;
        *step_delay = cdelay;
        LATGCLR = LED1 — LED2;
        LATGSET=LED1;
        break;
        case BTN2:
        *dir = 0;
        *mode = 0;
        *step_delay = cdelay;
        LATGCLR = LED1 — LED2;
        LATGSET=LED2;
        break;
        case 0x000000C0:
        *dir = 0;
        *mode = 1;
        *step_delay = cdelay;
        LATGCLR = LED1 — LED2;
        LATGSET = LED1 — LED2;
        break;
        default:
        *dir = 1;
        *mode = 1;
        *step_delay = cdelay;
        LATGCLR = LED1 — LED2;
                break;
        }
    }
```

The board used in the lab had a very troublesome button that required writing to the LEDs to guarantee it was actually pressed to differentiate hardware difficulties and problems with the FSM.

Speaking of the FSM, the next function dictates the next state behavior of the FSM and is most likely the most interesting part of the program:

which button and which state?

Listing 3.4.1

```
int stepperState(int dir  int mode) {
        sui pstate;
        if(dir == 1) {
                if (mode ==1) {
                        switch(pstate) {
                                case 6:
                                pstate = 0;
                                break;
                                case 7:
                                pstate = 1;
                                break;
                                default:
                                pstate+=2;
                                break;
                                }
                        }
                else {
                        switch(pstate) {
                                case 7:
                                pstate = 0;
                                break;
                                default:
                                pstate++;
                                break;
                                }
                        }
                }
```

Listing 3.4.2

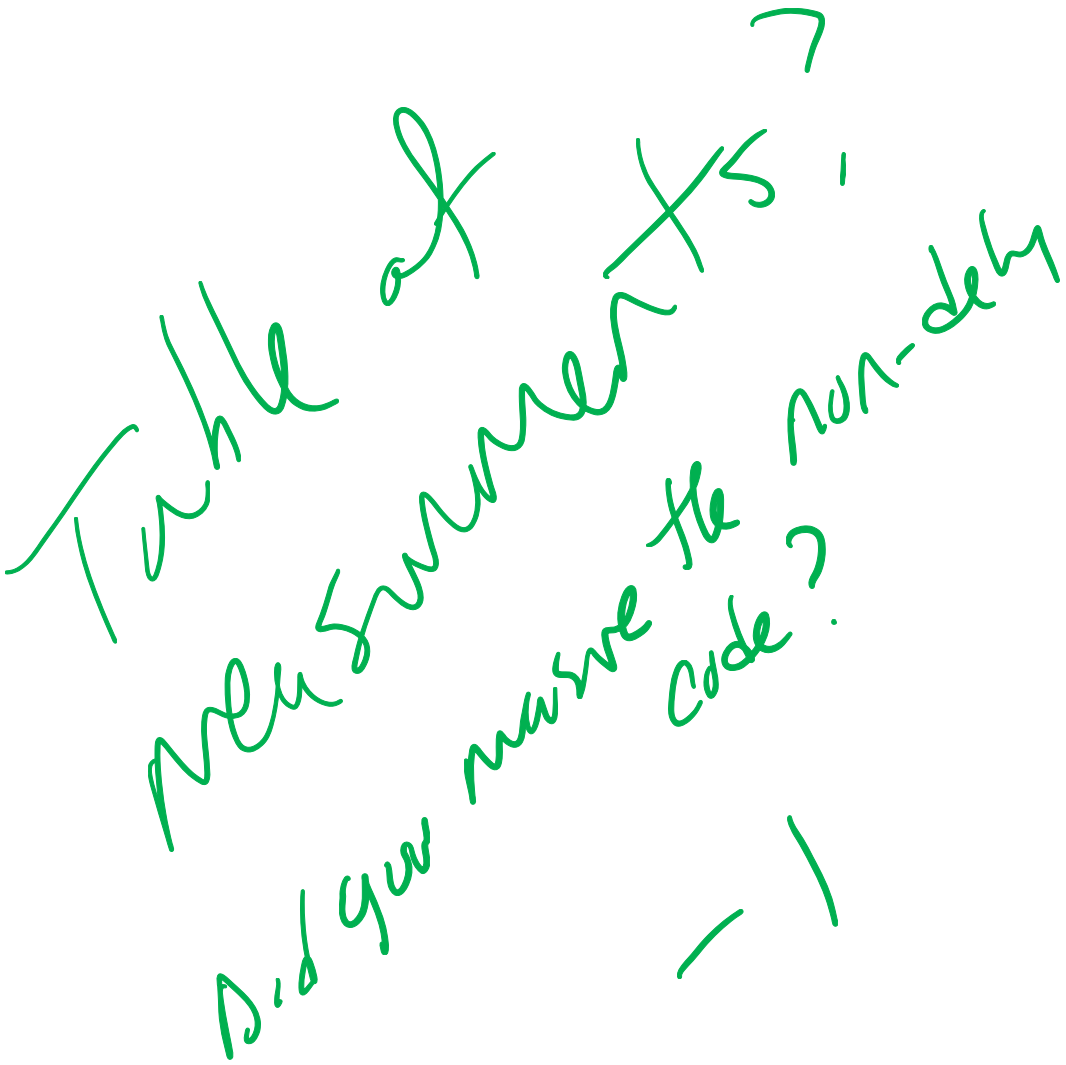
```
/* Direction Switch*/          else {
    if (mode ==1) {
        switch(pstate) {
            case 1:
            pstate = 7;
            break;
            case 0:
            pstate = 6;
            break;
            default:
            pstate=2;
            break;
            }
        }
    else {
        switch(pstate) {
            case 0:
            pstate = 7;
            break;
            default:
            pstate-;
            break;
            }
        }
    }
if (pstate > 7) {
    pstate = 0;
    }
if (pstate < 0) {
    pstate = 7;
    }
return pstate;
}
```

The above listing was split into 2 to keep it readable on the pages.

The last function simply just takes the state indicator and sends the signal to the stepper motor based on the state array from the header file

Listing 3.5

```
void stepperPush(int shoe) {
        LATBCLR = SM_COILS;
        uint write = stepshoe;
        write = write << 7;
        LATBSET = write;
        //LATBINV = stepshoe<< 2;
              }
```



3 Testing and Verification

For the testing, there was a lot of time spent watching the variables in the variable watch in the IDE. The main method of checking the functionality was of course pushing the buttons and watching the motor spin either in half or whole step mode.

To check to make sure that the timing was correct, the same data that was to be written to the motor was also written to the oscilloscope as well. The wave-forms of the data-write is shown below for each of the button configurations:

Figure 1:

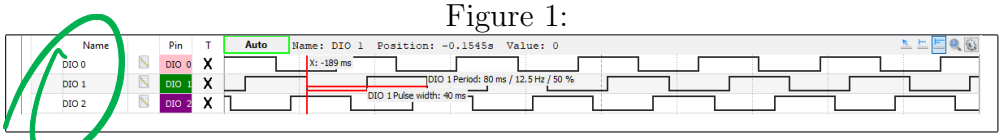


Figure 2:

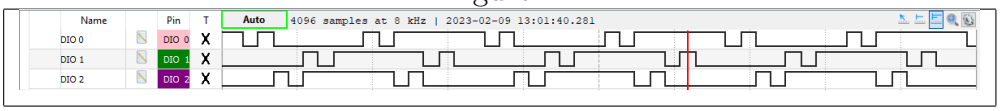


Figure 3:

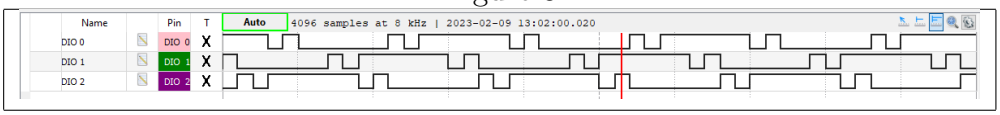
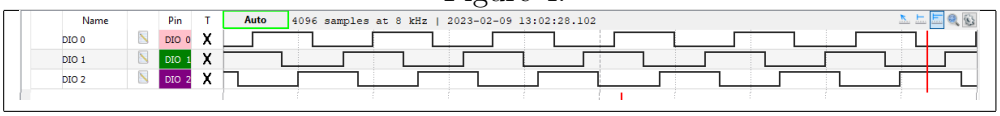


Figure 4:



4 Post Lab Questions

1. 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.

The different methods that I saw were either using switch statements to define the next state logic or an if else statements. The switch statement looks better in hindsight as it eliminates the possibility of erroneously fulfilling multiple state requirements and outputting the wrong state. The if else statement seemed at first to be better about handling cases where the state accidentally threw something outside the bounds of the FSM, but then I realized that the

default case should do something a little less controlled but similar enough when handling state assignments that were outside the bound of the state array.

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

Even though they had the same structure of read modify write, the microcontroller version was different as it continually looped as opposed to the functions waiting for the next set of instructions (not that the program or the CPU stopped, the functions of the machine simply did not advance to the next state without input). This lab had a continual polling and checking of inputs and states where as the other had a step to make sure the coast was clear before it ran again.

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

(and every step)

The buttons were sampled every iteration of the while loop, which means much more than it needs to be. The fix for this is the next weeks lab's concept of multi rate scheduling where the buttons are poled every x amount of while loop iterations, to be determined by how fast we want the routine to update.

polled

5 Conclusion

The lab was able to be complete using the delay function to change the speed of the motor. The speed was also able to be modulated by changing the steps per while cycle. The read modify write style of this lab was very familiar and was able to be implemented with relative ease.