Seth Cram

ECE341

9/21/2021

**Prelab 3: Stepper Motor with Software Finite State Machine**

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.

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.



