EENG461, LAB5 Report, Automatic Blinds

Mark Clark and Jeremy Munson

Objective

The objective of this lab is to read light levels from a photodiode and use the data to control a stepper motor as if adjusting blinds. The stepper motor is driven by an L293D quad half-H module, and the photodiode is read by an ADC.

Introduction

We are told two sets of three "tasks" that make up the requirements for the lab.

First task set:

- Task 1 Take a light measurement from a photodiode every second
- Task 2 Use it to control a stepper motor such that the brightness affects the amount that it is open in increments.
- Task 3 Every 10% change in brightness should cause one rotation of the stepper motor

Second task set:

- Task 1 Wire up photodiode to ADC and read it with a 1s timer, Conversion Math (to percentage of light range)
- Task 2 Printf the ADCV and the Percentage of light range
- Task 3 Implement the stepper motor in fixed movements such that you can control 1 exact rotation forward or backward (aka draw the rest of the owl) and use the percentage of light to determine how many to do

Narrative

We already have significant code and utilities prepared from the previous labs. Of note, we re-used existing code for ADC readings, printing over uart, and time-keeping. We call all of the various setup functions for each part, then enter the main loop.

The L293D module is a quad half-h-bridge driver with automatic dead-time insertion. It can be used completely transparently from our microcontroller, simply providing much higher current source/sink capability. Because there are 4 half-h inputs, we have just enoguh to drive our stepper motor. The V+ was connected to VBUS, and the four inputs were connected to PC4-PC7.

The stepper motor is… not of high quality or capability. Typical speeds of ~10rpm were achieved with 5v drive, with the motor seizing at speeds above that. The stepper motor was plugged into the driver module.

The photodiode was connected between 3.3v and gnd in series with a 480kohm resistor. The ADC pin was connected between the resistor and the photodiode.

Program Overview

We use one timer to perform ADC conversions on a regular basis, making the ADC data available in a global variable. We use another timer to keep track of the system "uptime", allowing the main loop to perform updates after every second. Finally, one more timer calls an interrupt regularly to control the stepper motor. If the stepper motor is not yet at it's intended location, then the step sequence will be advanced.

Main loop

`uptime_seconds` is updated by a timer each second, and indicates the number of seconds since the time-keeping utility started. At the start of our main loop we do nothing until at least a second has passed since the start of the last loop - this meets the 1-second response requirement of Task 1.

We then read in the current raw ADC value for the `light`. `light` is updated regularly by our existing ADC code, and could be updated mid-loop - which would make the calculations inconsistent. The read into temp prevents this issue.

Next, we dynamically adjust the minimum and maximum ranges for the reading of the light level - a new highest level or a new lowest level will expand the range of the measurement. This allows the program to adjust to different lighting in different rooms, for example. In practice, we were always able to use the entire range of <10% to 100% (in 10% increments) specified in Task 3.

We then calculate what position, 1 to 10, linearly from the light value. The `target_position` is in the range of 1 to 10, indicating the number of rotations to make relative to 0 rotations.

We print out the information using the printlf() function that we wrote in a previous lab. This meets the requirement of task 2 in the second set.

Finally, we call the calculateRotation() function, which starts the stepper motor movement if needed.

```
while (1) {
        while(last_time == uptime_seconds) {};
        last_time = uptime_seconds;

        int temp = light;
        if(temp < adc_lowest) adc_lowest = temp; //If this is a new lowest value
         ↪  ever seen, set a new floor
        if(temp > adc_highest) adc_highest = temp;

        int target_position = ((temp - adc_lowest)*100/(adc_highest-adc_lowest) +2)
         ↪  / 10;

        printlf("Target Position: %d \t | Light %d0% | raw adc: %d\n",
 ↪  target_position, target_position, temp);

        calculateRotation(target_position*360.0f);
    }
```

This function uses floating point because the "general" solution is going to be used in a project where floating point is going to make more sense than integers.

```
void calculateRotation(float angle) {
    stepper_motor_1.pending_steps += ((angle - stepper_motor_1.current_angle) /
 ↪  stepper_motor_1.stride_angle);
```

```
    stepper_motor_1.current_angle = angle;
}
```

Stepper Motor Operation

The stepper motor code was more complex than really needed for this lab, specifically. One of the team members intends to re-use it in their final project, so is working on making a more "general" solution for arbitrary stepper motors using arbitrary timers.

The stepper motor driver is connected to pins PC4-PC7. This allows writing the stepper motor state to occur at the same time; otherwise there could be glitching caused by a brief invalid combination of outputs. Because this is not in the "low" pins of the port, we left-shift the stepper motor state values to be written by 4 - aligning it with the correct pins.

The stepper motor operates by switching through a specific series of states - we chose the half-stepping order of states. Instead of using an array and index, we used a doubly-circle-linked list of valid states. This made advancing the stepper motor forward or backwards simply a matter of traversing forward or backwards in the linked list. This trades a small bit of memory (two additional pointers per state) for faster execution time - indexing an array with an (index % size) can be many more clock cycles, since there is no hardware modulo available. Using a branch at the top of the list is also potentially slower because of the pipeline flush. (A cost of 2 + pipeline instructions).

```
if (stepper_motor_1.pending_steps > 0) {
        GPIO_PORTC_DATA_BITS_R[stepper_motor_1.gpio_val] =
 ↪  stepper_motor_1.current_step->value << 4;
        stepper_motor_1.current_step = stepper_motor_1.current_step->next;
        stepper_motor_1.pending_steps--;
    } else if (stepper_motor_1.pending_steps < 0) {
        GPIO_PORTC_DATA_BITS_R[stepper_motor_1.gpio_val] =
 ↪  stepper_motor_1.current_step->value << 4;
        stepper_motor_1.current_step = stepper_motor_1.current_step->previous;
        stepper_motor_1.pending_steps++;
    } else {
        GPIO_PORTC_DATA_BITS_R[stepper_motor_1.gpio_val] = 0x0; //Turn coils off
 ↪  when move is completed
    }
```

Photodiode Operation

We changed the name of the variable for the ADC to "light". This was the only change from the ADC code used in prior labs. Tuning the response to the raw ADC value and calibrating on the fly is explained in the "main loop" section above.

Concluding Remarks

The lab was completed, accomplishing all tasks specified. It was interesting using the photodiode and stepper motor. The stepper motor is exceptionally slow, which made testing "painful". We might suggest that instead of one rotation per 10% light, we should maybe do 1/4 rotation per 10% because of the slowness of the motor.