Module Three Assignment: Optical Tachometer

Nicolas Miller

09/29/2024

1 Requirements

As specified in the assignment document, this project has been design to satisfy the following requirements:

- Implement an optical tachometer using an Arduino to measure the speed of a brushless propeller.
- Use an infrared emitter and detector pair to detect the rotation of the propeller.
- The propeller must be controlled and driven by the Arduino using appropriate motor control logic.
- Implement the system using either:
  - Round Robin with Interrupts
  - Function Queue Scheduling
- Measure and capture the propeller's revolutions per minute (RPM) over time using the IR emitter/detector data.
- Record the RPM measurements over some period of time.
- Transfer the recorded RPM data to a host computer for further analysis.
- Create a graph of the RPMs over time using the collected data.

2 Design

I spent a considerable time on the hardware design for this project. First, I selected hardware. After settling on an IR emitter/receiver pair and motor, I designed a 3D model for the project frame. The frame is composed of three columns where the IR emitter/receiver pair sit on the left and right columns with the motor and propellor sitting between them in the middle column. Additionally, I decided to add a status LED and a button. The status LED helped in testing and the button is simply used to start and stop the system.

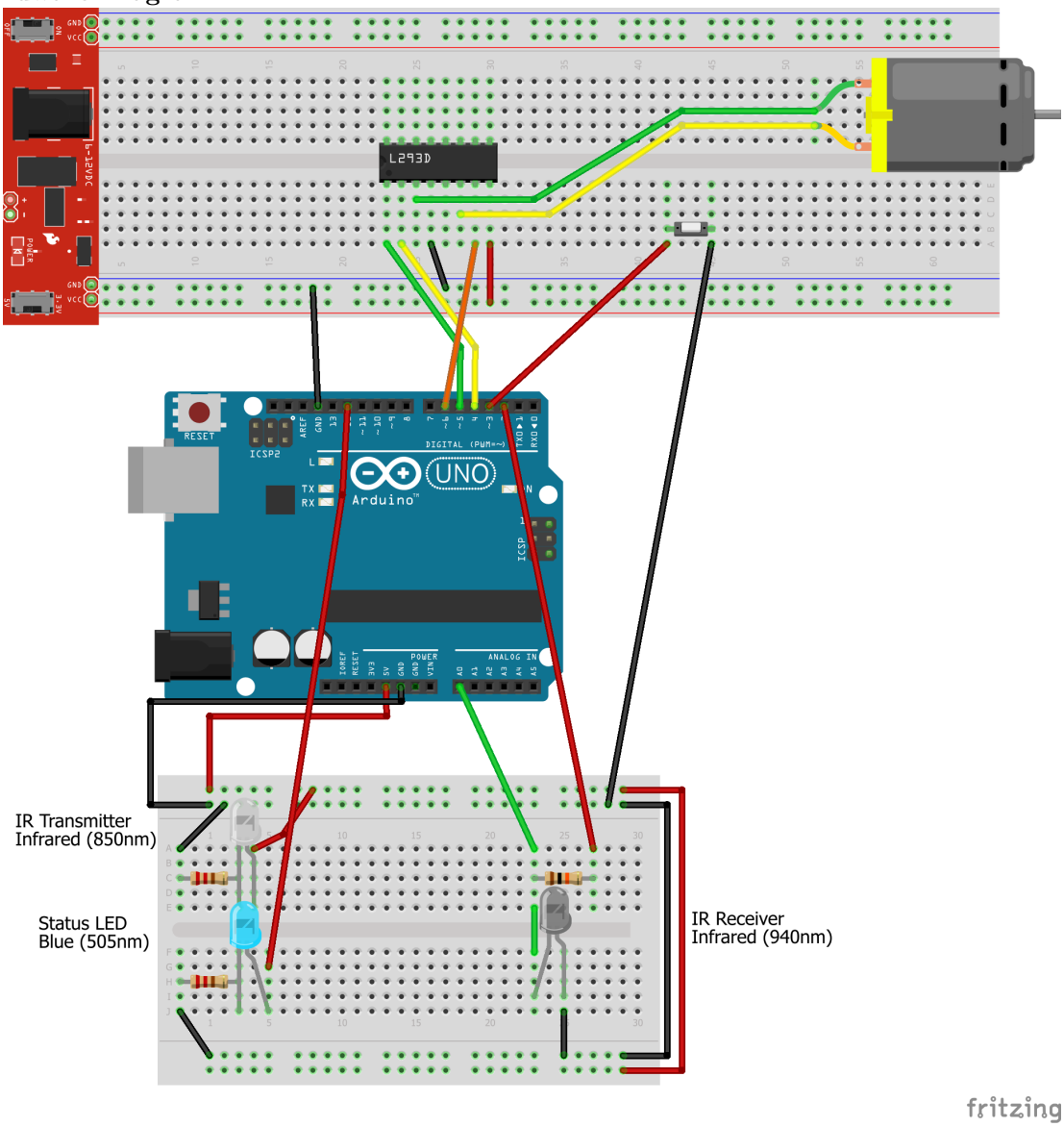
Having never worked with a motor before, I had to spend a bit of time learning how it should be driven and interact with the Arduino. The motor is an Adafruit hobby motor, and I use a separate power supply to provide power, while the Arduino transmits speed and direction information from a PWM pin and a digital pin. This is achieved using an L293D motor driver, which simplified the process of connecting and controlling the motor.

Regarding software, this project gave me much more trouble than the previous one. Given that my overall design includes only two hardware interrupts and one calculation function, I felt that a Round Robin with Interrupts approach was sufficient, as opposed to using a function queue scheduling architecture.

My program calculates RPM once per second and achieves this using the same hardware timer interrupt strategy that we used on the last project. To track rotations, I use a hardware timer where a variable that stores the number of rotations is incremented every time the IR emitter/receiver connection is broken. After one second, the `calculateRPM` function is called, which disables global interrupts and calculates the RPM. I disabled global interrupts during the calculation because the IR interrupt increments the rotation count, which is used to calculate RPM.

I did try adding a third button interrupt to the project, which was used to start and stop the system. However, I had a lot of trouble with it. After implementation, my system suffered from numerous false positive triggers, even with extreme debouncing timings. In the end, I simply poll the button in the main loop, starting and stopping the system as the button is pressed. I think this problem stemmed from electrical interference coming from the wires that powered the motor, which sat in close proximity to the button, and based on my research, it seems that pins configured as hardware interrupts are more sensitive than pins configured as digital inputs. I'm not sure how true that is, but it seemed to explain this problem and help me resolve it.

Hardware Diagram



3 Source Code

main.cpp

```
#include <Arduino.h>

// Pin Definitions
#define PD_PIN 2          // IR receiver power control pin
#define LED 12            // Status LED pin
#define sensorRead A0     // IR receiver output pin
#define ENABLE 5          // Motor enable pin (PWM control)
#define DIRA 4            // Motor direction pin A
#define DIRB 6            // Motor direction pin B
#define BUTTON_PIN 3      // Button pin for start/stop control

// Variables
volatile unsigned int rotations = 0;       // Counter for blade passes
volatile bool bladeDetected = false;       // Flag for blade detection
volatile bool calculateFlag = false;       // Flag for RPM calculation
bool systemRunning = false;                // Flag to check if the system is runn

const unsigned long rpmInterval = 1000;    // Interval for RPM calculation (1 sec
const unsigned long numBlades = 3;         // Number of blades on the fan
const unsigned long oneMinute = 60000;     // One minute in milliseconds
const int limit = 850;                     // IR detection threshold
unsigned long lastRPMTime = 0;             // Time of last RPM calculation

volatile unsigned long lastDebounceTime = 0; // Time of the last button press
const unsigned long debounceDelay = 200;     // Minimum debounce time (in millis

// Declare functions
void calculateRPM();
void irISR();

void setup()
{
  // LED and IR setup
  pinMode(PD_PIN, INPUT);       // IR receiver input pin
  pinMode(LED, OUTPUT);         // Status LED pin
  digitalWrite(PD_PIN, HIGH); // IR receiver powered on
  digitalWrite(LED, LOW);       // Status LED off initially

  // Motor setup
  pinMode(ENABLE, OUTPUT);      // Motor enable pin
  pinMode(DIRA, OUTPUT);        // Motor direction pin A
  pinMode(DIRB, OUTPUT);        // Motor direction pin B
```

```
43    digitalWrite(DIRA, LOW);    // Set motor direction A
44    digitalWrite(DIRB, HIGH);     // Set motor direction B
45
46    // Button setup
47    pinMode(BUTTON_PIN, INPUT_PULLUP); // Enable internal pull-up resistor
48
49    // Attach interrupt for the IR receiver pin (PD_PIN)
50    attachInterrupt(digitalPinToInterrupt(PD_PIN), irISR, FALLING);
51
52    // Serial setup
53    Serial.begin(9600);
54
55    // Set up an internal 1s timer interrupt. Used Arduino interrupts timer
56    // calculator tool deepbluembedded website to generate values and code.
57    cli();                  // Disable global interrupts
58    TCCR1A = 0;             // Set TCCR1A register to 0
59    TCCR1B = 0;             // Set TCCR1B register to 0
60    TCCR1B |= B00000100;    // Prescaler = 256
61    OCR1A = 62500;          // Timer Compare1A Register
62    TIMSK1 |= B00000010;    // Enable Timer COMPA Interrupt
63    sei();                  // Enable global interrupts
64  }
65
66  void loop()
67  {
68    // Read the button state
69    bool buttonState = digitalRead(BUTTON_PIN);
70
71    // Check for button press (active LOW)
72    if (buttonState == LOW && (millis() - lastDebounceTime) > debounceDelay)
73    {
74      lastDebounceTime = millis(); // Update debounce timer
75
76      // Toggle system running state
77      systemRunning = !systemRunning;
78    }
79
80    if (systemRunning)
81    {
82      // System is running, start motor at full speed and process RPM
83      analogWrite(ENABLE, 255);
84
85      // Flash Status LED on Blade Detection
86      if (bladeDetected)
87      {
88        bladeDetected = false; // Reset flag
```

```
89
90          // Flash the LED
91          digitalWrite(LED, HIGH);
92          delay(100);
93          digitalWrite(LED, LOW);
94        }
95
96        // RPM Calculation Every Second
97        if (calculateFlag)
98        {
99          calculateFlag = false;       // Reset flag
100         calculateRPM();              // Calculate RPM
101       }
102     }
103     else
104     {
105       analogWrite(ENABLE, 0); // Stop motor
106       rotations = 0;          // Reset rotations
107     }
108   }
109
110   // Interrupt Service Routine for Timer1
111   ISR(TIMER1_COMPA_vect)
112   {
113     OCR1A += 62500;
114     calculateFlag = true; // Set flag for RPM calculation
115   }
116
117   // ISR is called when the IR beam is interrupted
118   void irISR()
119   {
120     rotations++;          // Increment rotations
121     bladeDetected = true; // Set flag for blade detection
122   }
123
124   // RPM Calculation Function
125   void calculateRPM()
126   {
127     cli(); // Disable global interrupts during calculation
128     float rpm = (rotations / numBlades) * (oneMinute / rpmInterval); // Calculate
129     sei(); // Enable global interrupts after calculation
130
131     // Print time and RPM
132     Serial.print(millis());
133     Serial.print(",");
134     Serial.println(rpm);
```

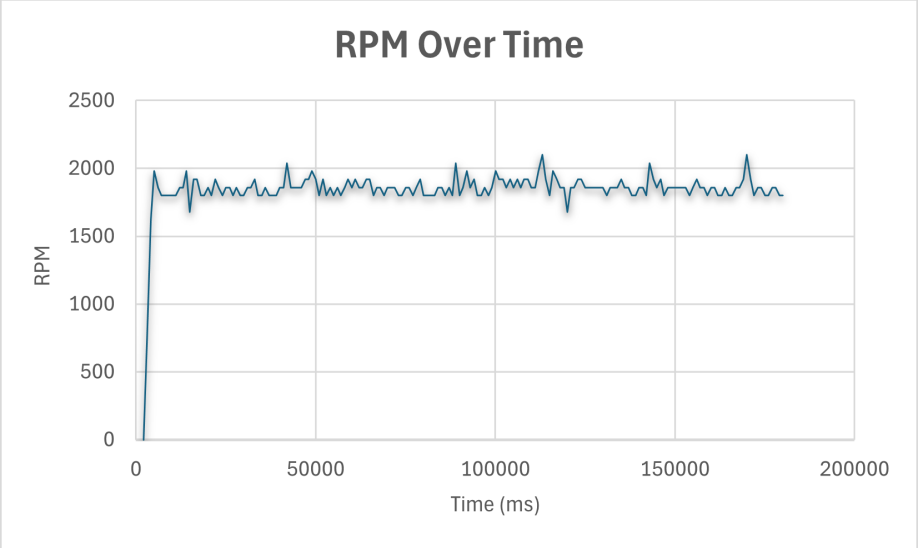
```
135
136     rotations = 0; // Reset blade counter for the next interval
137   }
```

4 Video

A video demonstration of the project can be found at the following link: https://youtu.be/EF5tFeT3evU

5 Data

Here are the results of the RPM measurements:



In this graph, we see a lot of variability, but the average RPM values fell between 1800 and 1860 based on my observations. The spikes and dips can likely be attributed to the 9V battery that I used as a power source and the fact that the motor is undervolted. Additionally, the weight of the electrical tape on the fan blades and the possibility of false positives or missed readings from the IR emitter/receiver pair could also account for those observed spikes and dips.

6 Sharing Statement

I will share this entire PDF submission in the sharing discussion board.