EN.605.715.81.FA25 Project 3 - Arduino RT RPM Tachometer

Kelly “Scott” Sims

Github Link:
https://github.com/Mooseburger1/johns_hopkins_masters/tree/main/software_development_for_rt_embedded_systems_EN_605_715_81/project_3/propeller_speed/src

YouTube Link: https://youtu.be/Dpa_Z_6-zlo

Requirements

Project Key Requirements

1. Using Function Queue scheduling, measure RPM of brushless DC motor
2. Transmit the values across a Serial bus like USB to your Host, others could be SPI or I2C if your Host had one of those buses.
3. Export as a comma separated value file, read into a spreadsheet program and plot the RPM vs time

Requirement Alterations

1. Consistent
2. Data will be transferred via WiFi over UDP.
3. Data will be streamed and plotted in real time from the UDP client that establishes connection with the microcontroller

Hardware & Protocol Considerations

Microcontroller: renesas-ra platform (E.g. Arduino R4 Uno)
Communication: WiFi
Transport Protocol: UDP
Sensor: IR LED, IR PhotoTransistor
Hardware: Brushless DC Motor

Design

WiFi Connection

WiFi credentials are statically persisted in configuration header as global constants. This allows them to be utilized anywhere throughout the codebase. These credentials are the SSID of the network and its password.

A WiFi Setup library is created that consumes these credentials via a configuration struct, attempts to connect, and reports the IP address of the microcontroller. The configuration struct is the following

```
struct WiFiConfigurations {
  const char* ssid;
  const char* password;
  const StartUpMessage* startup_message;
  const ConnectedMessage* connected_message;
};
```

Where startup_message and connected_message are user defined messages to be displayed when initializing WiFi connection

UDP Connection

After WiFi connection is established, a UDP listener will be initialized. It will listen for and transmit data to a client over port 1234

Clock Sync

The RTC library is utilized to sync the clock on the microcontroller. To keep the dependencies light, the microcontroller will not be responsible for setting the time. It is the client's responsibility to provide the unix timestamp as the first UDP packet to the microcontroller.

It is from this unix timestamp that subsequent data transmissions will reference.

Client Ready

Once the clock is synced, an ISR function will be attached on digital pin 3 and be triggered by a rising value. This ISR will simply count the number of times the blade has broken the IR beam from the IR receiver.

Function Queue

A function queue will hold two primary functions, CalculateRPM & TransmitRPM. The CalculateRPM function utilizes the blade count incremented by the ISR. Prior to calculating, it creates a copy of the current blade count while disabling interrupts. It persists the current RPM value in a global volatile variable.

Once the TransmitRPM function is executed, it stringifies the current RPM value persisted by CalculateRPM. This is safe to do so without synchronization on the global variable. This is because the CalculateRPM and TransmitRPM can not be run asynchronously, meaning there's no risk of a race condition.

UDP Client

The UDP client is a simple python client that connects to the microcontroller and transmits the unix timestamp.

Once connected the client starts two background threads
1. One for receiving data across the UDP connection
2. One for listening for user inputs to terminate the connection

This leaves the main thread free to plot, update, and render the real-time graph when new data arrives.

Wiring

IR LED
The IR LED is a simple circuit and only has a single purpose of constantly emitting IR light.

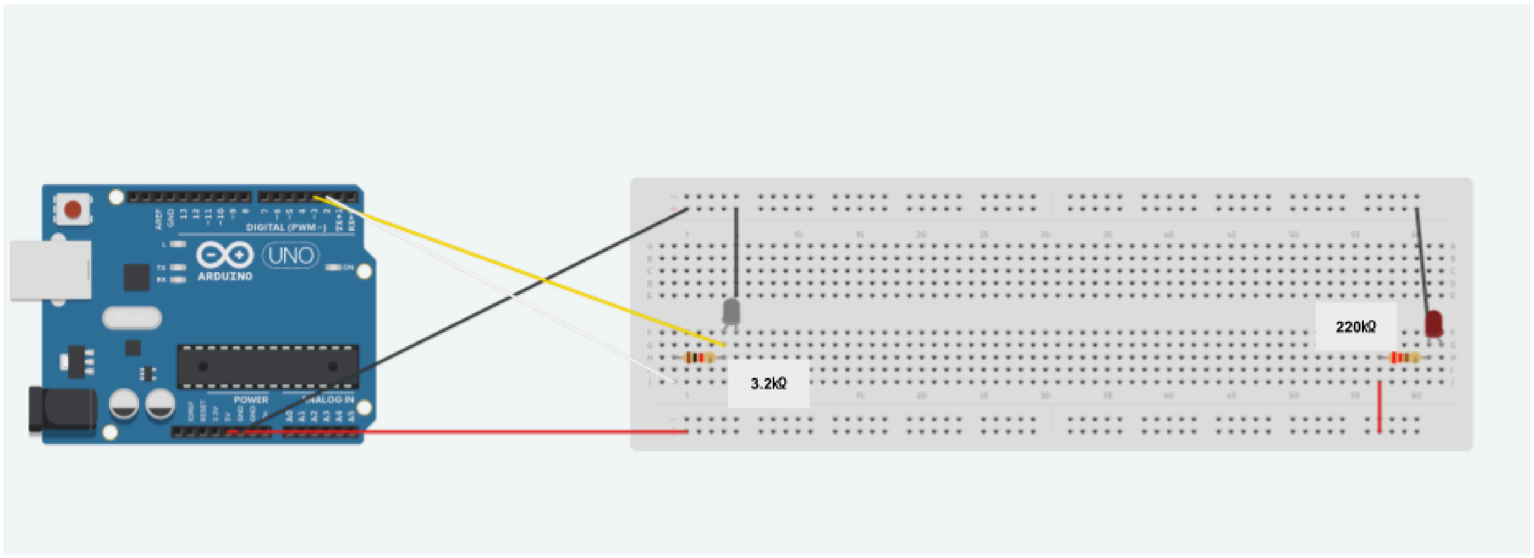
- **5V -> 220Ω -> IR LED -> Ground**
    - Simple wiring of 5v pin from Arduino to a 220 ohm resistor. The resistor is in series with the IR LED which completes the circuit to ground

PHOTOTRANSISTOR
The Phototransistor is configured in a reverse bias orientation with 5V. When no IR light is incident on it, it prevents current from flowing. Hence all 5v is routed to pin 3 on the arduino board. When IR light interacts with the phototransistor, it allows current to flow. The amount of current is directly proportional to the amount of incident IR light. As more IR light is received, more current flows through the phototransistor to ground. Hence the voltage seen at pin 3 will decrease significantly.

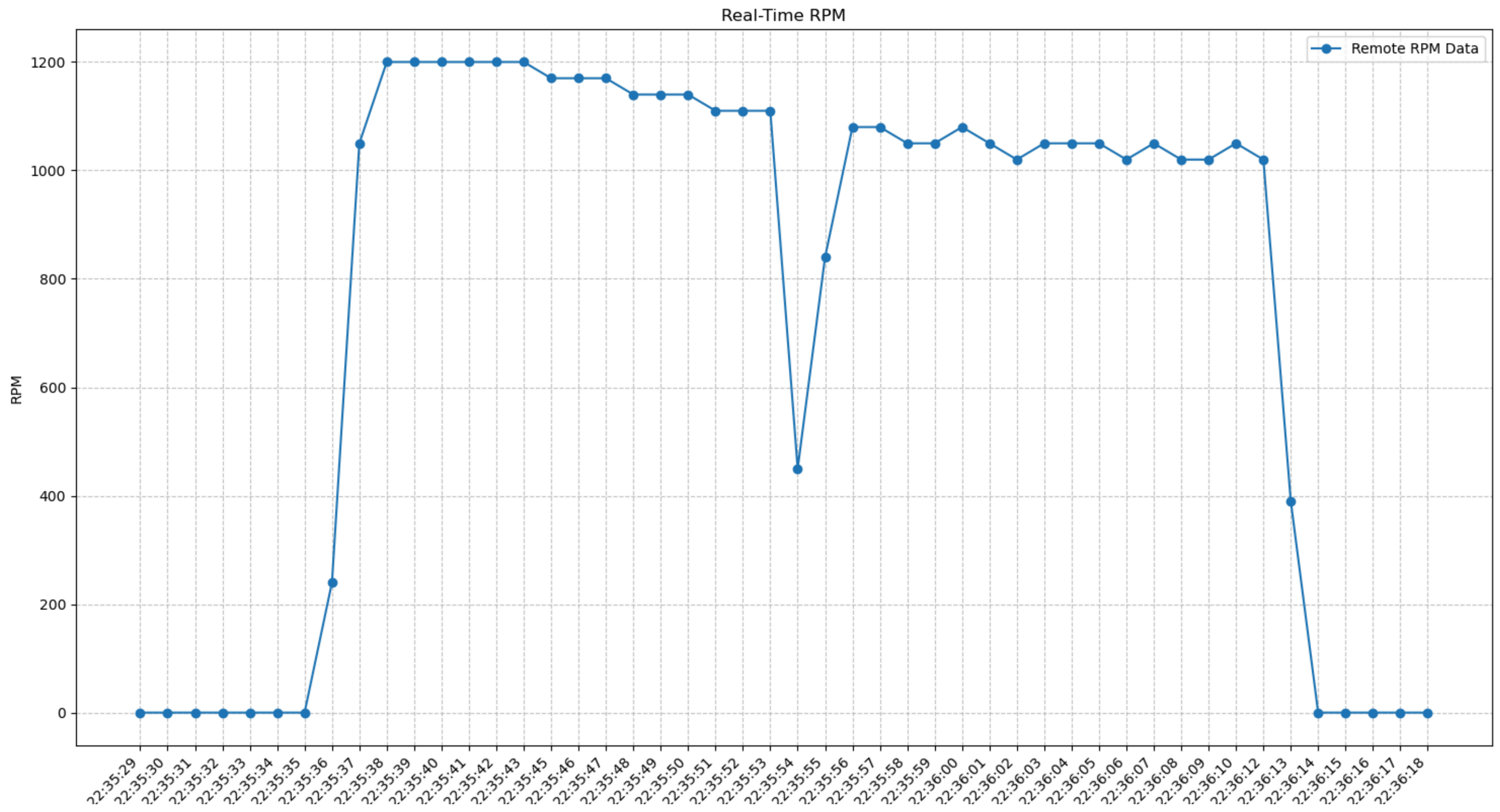
PhotoTransistor -> Ground
- **5V -> 3.2 kOhm resistor <**

Digital Pin 3

- Using digital pin 2 as a 5v power supply to 3.2 kOhm resistor. From the resistor, it makes two branches. One to digital pin 3 of the Arduino. The other to the PhotoTransistor which completes the circuit to ground



Implementation



```cpp
// main.cpp

#include <Arduino.h>
#include <WiFiS3.h>
#include <WiFiUdp.h>

#include "configurations.h"
#include "rtc_config.h"

// Pin used exclusively to be a 5V power supply for the phototransistor
const int PD_POWER_PIN = 2;
// Pin to read high / low when the propeller breaks the IR beam
const int SENSOR_PIN = 3;
const int PROPELLER_BLADES = 2;

const unsigned long RPM_CALCULATION_INTERVAL_MS = 1000;
const unsigned long TRANSMIT_INTERVAL_MS = 1000;
const unsigned long STOPPED_THRESHOLD_MS = 2000;

volatile unsigned long bladePassCount = 0;
volatile unsigned long lastBladePassTime = 0;

volatile int ready_to_transmit = 0;
float currentRpm = 0.0;
unsigned long lastCalcTime = 0;

WiFiUDP udp;

using TaskFunction = void (*)();

struct Task{
    TaskFunction function;
    unsigned long interval;
    unsigned long lastExecuted;
};

void calculateRPM();
void transmitRPM();

Task taskQueue[] = {
    {calculateRPM, RPM_CALCULATION_INTERVAL_MS, 0},
    {transmitRPM, TRANSMIT_INTERVAL_MS, 0},
};

const int numTasks = sizeof(taskQueue) / sizeof(Task);

void CountBladePassIsrFunction() {
    ++bladePassCount;
    lastBladePassTime = millis();
}

void calculateRPM() {
    unsigned long currentTime = millis();
    unsigned long elapsedTime = currentTime - lastCalcTime;

    noInterrupts();
    unsigned long count = bladePassCount;
    unsigned long lastPassTime = lastBladePassTime;
    bladePassCount = 0;
    interrupts();

    if (lastPassTime != 0 && (currentTime - lastPassTime) >= STOPPED_THRESHOLD_MS) {
        currentRpm = 0.0;
        lastCalcTime = currentTime;
        return;
    }


    if (elapsedTime > 0) {
        float revolutions = (float)count / PROPELLER_BLADES;
        currentRpm = (revolutions / (elapsedTime / 1000.0)) * 60.0;
        lastCalcTime = currentTime;
    }
}

void transmitRPM() {
    RTCTime curr_time;
    RTC.getTime(curr_time);
    auto payload = curr_time.toString() + ", " + String(currentRpm, 2);

    udp.beginPacket(udp.remoteIP(), udp.remotePort());
    udp.print(payload);
    udp.endPacket();

    Serial.println(payload);
}

// The runScheduler loops through all tasks in the queue and executes them if their last executed
// time is greater than or equal to their expected scheduled interval. The tasks are not dequeued from
// the queue. Instead their last execution time is persisted within and checked at each execution.
// This removes the need to constantly enqueue the same tasks when it will only ever be these two.
void runScheduluer() {
    unsigned long now = millis();
    for (int i = 0; i < numTasks; ++i) {
        if (now - taskQueue[i].lastExecuted >= taskQueue[i].interval) {
            taskQueue[i].function();
            taskQueue[i].lastExecuted = now;
        }
    }
}

void setup() {
    Serial.begin(9600);

    pinMode(PD_POWER_PIN, OUTPUT);
    digitalWrite(PD_POWER_PIN, HIGH);

    pinMode(SENSOR_PIN, INPUT);


    while (wifi_configs::ConnectToWiFi(wifi_configs::SSID, wifi_configs::PWD)) {
        Serial.println("Unable to connect to Wifi..."
        "You are currently trapped in an infinite loop!!!");
    }

    udp.begin(wifi_configs::PORT);
    Serial.print("UDP Server started on port: ");
    Serial.println(wifi_configs::PORT);

    while(!ready_to_transmit) {
        config::WaitForClockConfiguration(udp, [](){
```

```
        ready_to_transmit = 1;
        });
    }

    attachInterrupt(digitalPinToInterrupt(SENSOR_PIN), CountBladePassIsrFunction, RISING);

    lastCalcTime = millis();
}

void loop() {
    runScheduluer();
}
```

```
// configurations.h

#ifndef CONFIGURATIONS_H
#define CONFIGURATIONS_H

#include <Arduino.h>

namespace wifi_configs {

inline constexpr char* SSID = "FibreBox_X6-421B57";
inline constexpr char* PWD = "Enter Your Password Here";
inline constexpr int PORT = 12345;

inline int ConnectToWiFi(const char* ssid = SSID, const char* password = PWD) {
    if (!WiFi.begin(ssid, password)) {
        return 1;
    }

    while(WiFi.status() != WL_CONNECTED && WiFi.status() != WL_CONNECT_FAILED) {
        delay(100);
        Serial.print(".");
    }

    if (WiFi.status() == WL_CONNECT_FAILED) {
        Serial.println("Failed to connect to WiFi!");
        return 1;
    }

    Serial.print("Arduino WiFi IP Address: ");
    Serial.println(WiFi.localIP());

    return 0;
};

}  // namespace wifi_configs

#endif CONFIGURATIONS_H
```

```
// rtc_config.h


#ifndef RTC_CONFIG_H
#define RTC_CONFIG_H

#include <Arduino.h>
#include <WiFiS3.h>

#include "RTC.h"

namespace config {
using Callback = void(*)();

time_t ParseTimeFromUdp(const char* packet) {
  char* end_ptr;
  unsigned long epoch = strtoul(packet, &end_ptr, 10);
  Serial.print("Converted unix timestamp: ");
  Serial.println(epoch);

  if (*end_ptr != '\0') {
    Serial.println("Error: Non-numeric characters in timestamp");
    return 0;
  }

  if (epoch < 946684800UL || epoch > 4102444800UL) {
    // sanity check ~2000-01-01 to ~2100-01-01
    Serial.println("Error: Epoch timestamp out of valid range");
    return 0;
  }

  return (time_t)epoch;
}

void WaitForClockConfiguration(WiFiUDP& udp, Callback cb) {
    char udp_packet[256];

    Serial.println("Waiting for UDP Connection...");
    while (1) {
      if (udp.parsePacket()) {
        int dataLen = udp.available();
        udp.read(udp_packet, 255);
        udp_packet[dataLen] = '\0';
        Serial.print("Received epoch: ");
        Serial.println(udp_packet);

        time_t config_epoch = ParseTimeFromUdp(udp_packet);

        if (config_epoch != 0) {
          RTC.begin();
          RTCTime config_time_from_udp(config_epoch);
          RTC.setTime(config_time_from_udp);
          Serial.println("RTC time has been set.");

          udp.beginPacket(udp.remoteIP(), udp.remotePort());
          udp.print("OK");
          udp.endPacket();

          cb();

          return;
        }
      }
    }

    delay(500);
};

} // namespace config

#endif RTC_CONFIG_H
```

NOTE: The same UDP client code was used from project 2. The only difference is the titling on the plotted graph.