NAIT
Edmonton, Alberta

Remote Control Black Box

As a submisston to
Mr. Ross Taylor, Instructor
CNT Department

Submitted by
Michal Szmaj, Student
Computer Engineering Technology

April 21, 2017

Table of Contents

List of Figures and Tables v

Abstract vi

1.0 Introduction 1

2.0 Project Design 1

2.1 Features and Requirements 1

2.2 Tools 2

2.2.1 Arduino 2

2.2.2 Raspberry Pi 2

2.2.3 User Application 2

2.3 Components 3

2.3.1 ADXL335 4

2.3.2 BMP180 5

2.3.3 GPS Featherwing 6

2.3.4 Arduino Nano 7

2.3.5 Raspberry Pi Zero W 7

2.3.6 PowerBoost 1000C 8

2.3.7 Enclosure 9

2.4 Architecture 10

2.4.1 Arduino Code Architecture 11

2.4.2 Raspberry Pi Code Architecture 12

2.4.3 User Application Code Architecture 13

3.0 Application Implementation 14

3.1 Arduino Code Implementation 14

3.1.1 Accelerometer 14

3.1.2 Barometer 15

3.1.3 GPS 17

3.1.4 Main 18

3.1.5 SerialCommunication 19

3.2 Raspberry Pi Code Implementation 19

3.2.1 Parse 19

3.2.2 Database & Database Interface 20
3.2.3 Server...21
3.3 User Application Code Implementation 22
3.3.1 MainMenuView...22
3.3.2 ConnectView 22
3.3.3 UserDataView 23
3.3.4 LogTableView...23
3.3.5 Socket & ConnectionDelegate 24
3.3.6 JSONParser && RCBBData...25
3.3.7 Google Maps & Debugging 26
4.0 Project Results...27
5.0 Conclusion 27
6.0 Reflection..28
References 29

List of Figures and Tables

Figures

Figure 1. RC Black Box Schematic 3
Figure 2. ADXL335 Connection with Arduino Nano..4
Figure 3. BMP180 Connection with Arduino Nano 5
Figure 4. GPS Connection with Arduino Nano 6
Figure 5. Arduino Nano Connection with Raspberry Pi Zero W 7
Figure 6. Raspberry Pi, PowerBoost 1000C Connection 8
Figure 7. RC Black Box Enclosure 9
Figure 8. RC Black Box High Level Workflow 10
Figure 9. Arduino Code Architecture Diagram 11
Figure 10. Raspberry Pi Zero Code Architecture Diagram.......................................12
Figure 11. User Application Code Architecture Diagram 13

Tables

Table 1. BMP180 Calibration Values 16

Abstract

This report reviews the technical details regarding the development of the Remote Control Black Box. The RC Black Box is a small, light-weight device that attaches to any remote controlled vehicle. The device provides live information to the user via an *iPhone* application. The data includes X, Y, Z axes from an accelerometer, temperature, pressure, latitude, longitude, altitude, speed, GPS signal quality, and number of satellites the GPS is currently using. The device is constructed using several devices: an *ADXL335* for accelerometer data, a *BMP180* for barometer data, and an *Adafruit GPS Ultimate Featherwing* for GPS data. In order to collect the data an *Arduino Nano* is used and a *Raspberry Pi Zero W* is used to connect with the *iPhone* application.

1.0 Introduction

Piloting remote controlled planes, drones, and cars is a popular hobby which can potentially be very expensive. Many people buy RC vehicles that are already built and have software ready to work, while others build their own. In either situation, regardless of the amount of money spent on the RC vehicle, it can be useful to gather data from the device in order to become a better pilot, find a lost vehicle, determine why the vehicle crashed, or just out of interest. Many RC vehicles already come with hardware and software that can provide these metrics, but, a significant amount do not and this is where the RC Black Box can be useful.

The purpose of this report is to document the process of designing and producing a lightweight, physically small, and robust piece of hardware that can collect various data about the current flight and report back to the user. The goal is to have multiple sensors for data collection, a device for retrieving the data, and another for sending the data back to the user for analysis. The reason I have chosen to do this project is because it combines a significant amount of knowledge that has been taught during CNT and I'm free to explore different solutions in order to make this project work properly. That said, I also have been in similar situations where my drone was lost and I had to determine what went wrong and where to find the vehicle and this project can, hopefully, prevent future incidents.

2.0 Project Design

The following sections will document the project design. This includes the tools used, the components, and the code architecture. A lot of care was taken when designing the device in that everything was to be as modular as possible. Each physical component can be replaced and the software and the other hardware will work with the replacement. The code was also designed to be modular; a library could be replaced with another library with the same functionality and the device should still function. The modularity allows for future expansion.

2.1 Features and Requirements

The final product should be able to provide the user several pieces of data: X, Y, Z axes, temperature, pressure, altitude, latitude, longitude, speed, GPS quality, and number of satellites. This should all be displayed in an easy to use mobile application — this was initially specified to be a desktop application, however, it makes most sense that the data would be read in the field, rather than at home after the fact — and set up should be minimal as well. The user should also be able to view all of the data (or a subset). The device should be battery powered and they battery should last at least for two flights of an RC vehicle. Aside from the technological features of the device, it should also be light and compact as to not interfere with the RC vehicle.

2.2 Tools

The RC Black Box contains three main software components that utilise several different languages, frameworks, and programming tools. Each component used a different set of tools, languages, and frameworks and it documented below.

2.2.1 Arduino

The *Arduino* software was developed using the *PlatformIO* IDE which is just an add-on for the *Atom* text editor. *PlatformIO* allowed me to compile and load the code directly to the *Arduino* and view any serial communication. *PlatformIO* does not provide a hardware debugger and it was extremely difficult to debug. In order to debug I wrote a few debugging libraries and the code is littered with *print* function calls which are used to examine different values while the code is operating. Several standard C/C++ libraries (i.e. stdio.h, stdlib.h, math.h, etc.) were used.

2.2.2 Raspberry Pi

The *Raspberry Pi* runs the latest version of the *Raspbian Jessie* operating system and the software written for the *Pi* was written in *Python* in using the text-editor *Vim. Vim* is a console based text editor that is fairly easy to use, however, it does have a disadvantage in that it does not have a debugger — so, in order to debug, the *print* function was utilised.

2.2.3 User Application

The user application runs on *iOS* devices (iPhone, iPad, etc.), is written in the *Swift* programming language, and was developed using the IDE Xcode. Two *iOS* specific libraries were used: UIKit and Foundation. The *Google Maps* API was also used in order to display the current location of the device. Xcode does provide a way to debug code and it also provides an interface builder that was used to develop the user interface.

2.3 Components

This section contains a brief description of the devices and sensors that the RC Black Box is utilising and how they are connected. The 'Software' section below will detail how each device works with one another. Figure 1 below provides the schematic for the device. All figures in this section were designed and developed using the *Fritzing* application by Michal Szmaj (myself).

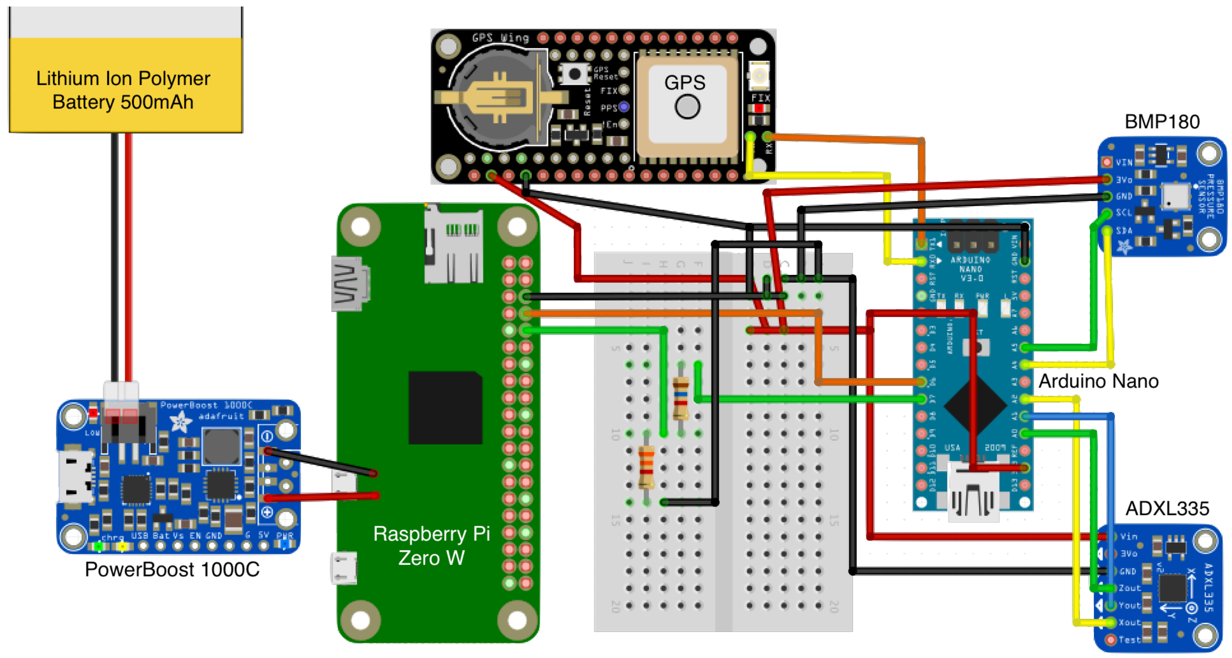

Figure 1. RC Black Box Schematic

2.3.1 ADXL335

"The ADXL335 is a small, thin, low power, complete 3-axis accelerometer with signal conditioned voltage outputs" (Analog Devices Inc, 2009, p. 1). It can measure *static acceleration* of gravity therefore allowing the device to sense its orientation on a 3D plane. The data can be used to determine if an RC vehicle is upright or not. This device was chosen because it can operate with an appropriate voltage for the *Arduino* (1.8V - 3.6V) and its size is relatively small (4mm x 4mm x 1.45mm) (Analog Devices Inc, 2009, p. 1). The device is read by wiring the 3V and ground to the appropriate pins on the *Arduino,* and the X, Y, and Z pins to *A0, A1,* and *A2* pins, respectively, on the *Arduino.* Figure 2 depicts the accelerometer connection with the *Arduino*.

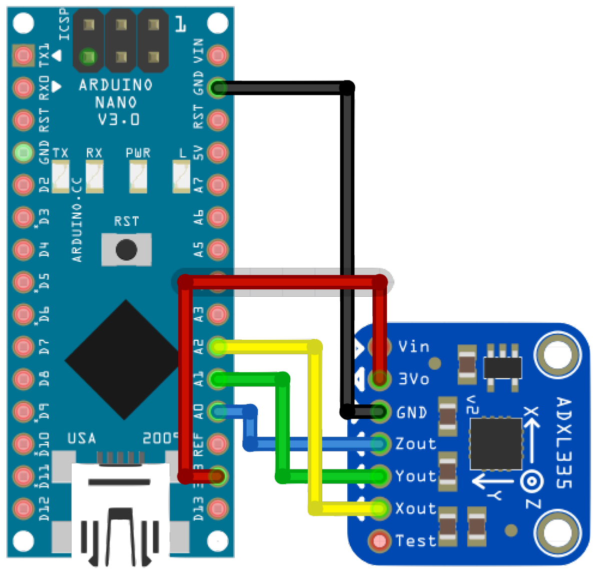

Figure 2. ADXL335 Connection with Arduino Nano

2.3.2 BMP180

The BMP180 is an “ultra-low power, low voltage” pressure sensor that is comparable in size (3.6mm x 3.8mm x 0.93mm) to the ADXL335 and allows quick reading of pressure and temperature (BOSCH, 2013, p. 2). The sensor is connected to the *Arduino* using I^{2}C, which allows the GPS to use hardware serial and the *Raspberry Pi* to use a software serial connection. This device was chosen because of its small size and low power consumption. Figure 3 depicts the barometer connection with the *Arduino*.

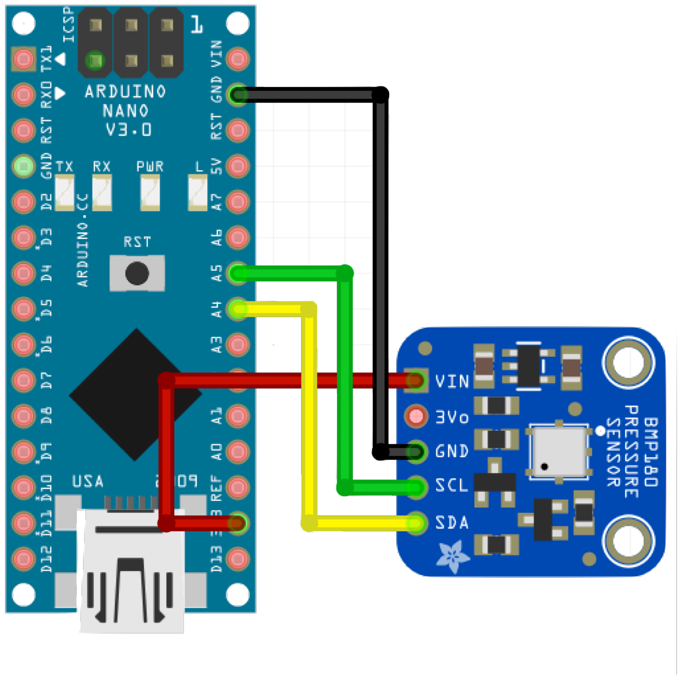

Figure 3. BMP180 Connection with Arduino Nano

2.3.3 GPS Featherwing

The *Adafruit Ultimate GPS Featherwing* breakout board is a standalone GPS module with ultra-high sensitivity, low power consumption (GlobalTop Tech Inc, 2011, p. 1), and a compact size (16 mm x 16 mm x 4.7mm). The board is connected to 3V and ground and is connected to the *Arduino*'s hardware serial pins (Rx and Tx). Figure 4 depicts the GPS connection with the *Arduino*.

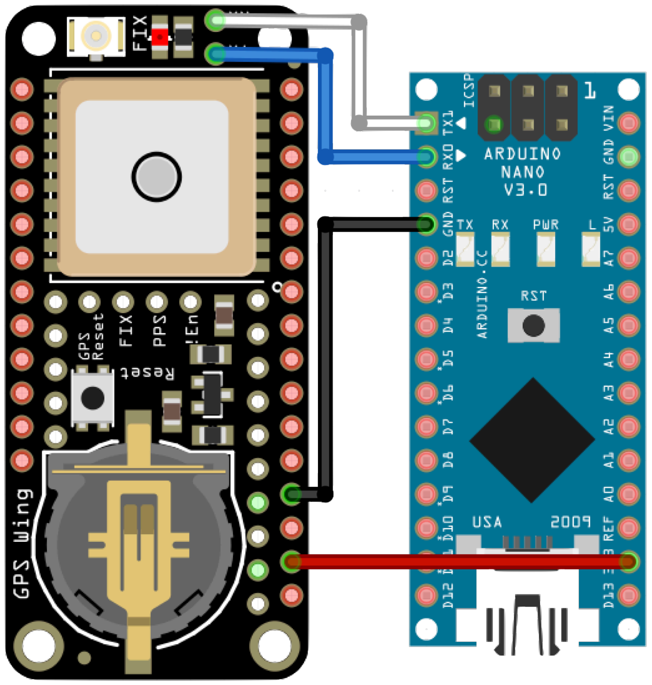

Figure 4. GPS Connection with Arduino Nano

2.3.4 Arduino Nano

The *Arduino Nano* is a single-board micro-controller that is significantly smaller than the *Arduino Uno* which was originally being used for the project — the *Uno* is 68.6mm x 53.4mm, while the *Nano* is 18mm x 45mm. The two boards operate identically and can run the same code unmodified — the only difference is in the compiler settings. This device was chosen for data collection because it is fairly easy to interface with each of the sensors, its small size, and because it does not run an operating system and it can utilise all of its resources for data collection; if the *Pi* was the data collection device there could be issues with precise data collection because it would have other operating system processes that could take precedence. Figure 5 depicts the connection between the *Arduino* and the *Pi*. A voltage divider is necessary in order to adjust for the 5V to 3.3V difference.

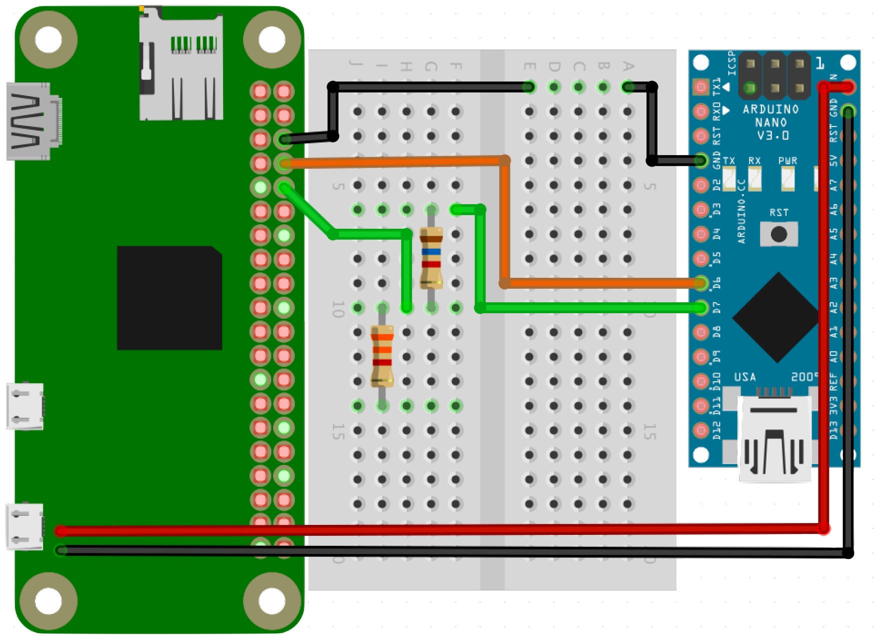

Figure 5. Arduino Nano Connection with Raspberry Pi Zero W

2.3.5 Raspberry Pi Zero W

The *Raspberry Pi Zero W* is a small, single-board computer with Wi-Fi and Bluetooth connectivity built in — this version of the device was chosen instead of the original *Raspberry Pi Zero* because of the wireless functionality. Because the data collection is done by the *Arduino,* the *Pi* can focus on more complicated tasks like parsing data, updating the database, and hosting the server.

2.3.6 PowerBoost 1000C

The *PowerBoost 1000C* is a small power supply that converts 3.7V to 5.2V from a Lithium Ion Polymer battery. The size of the device is comparable to the other components in the project at 36.2mm x 22.86mm. The device sits atop the *Raspberry Pi* and the positive and negative pins of the USB output are directly soldered to the *Pi*'s USB power input. This reduces bulk by not having a large USB cable connecting the two components.

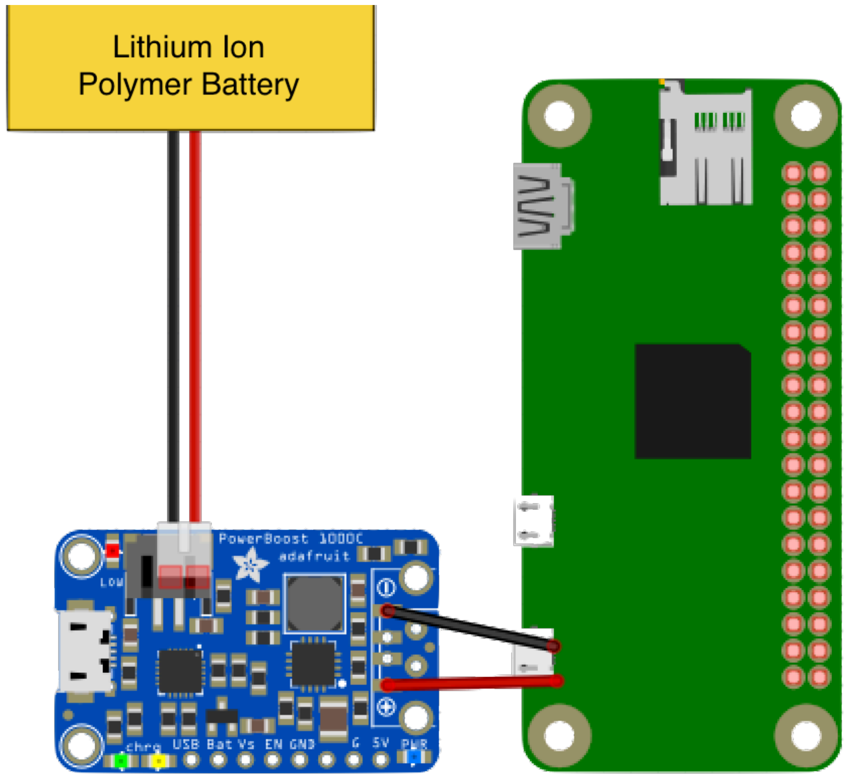

Figure 6. Raspberry Pi, PowerBoost 1000C Connection

2.3.7 Enclosure

The enclosure for the device is a 7.5cm x 8.5cm 3D printed case. The case is composed of two pieces that snap together. There are four holes in the enclosure: three for status lights (i.e. powered on, charging, low power) and one for the USB port and power switch (not pictured). Figure 7 showcases the 3D model of the enclosure.

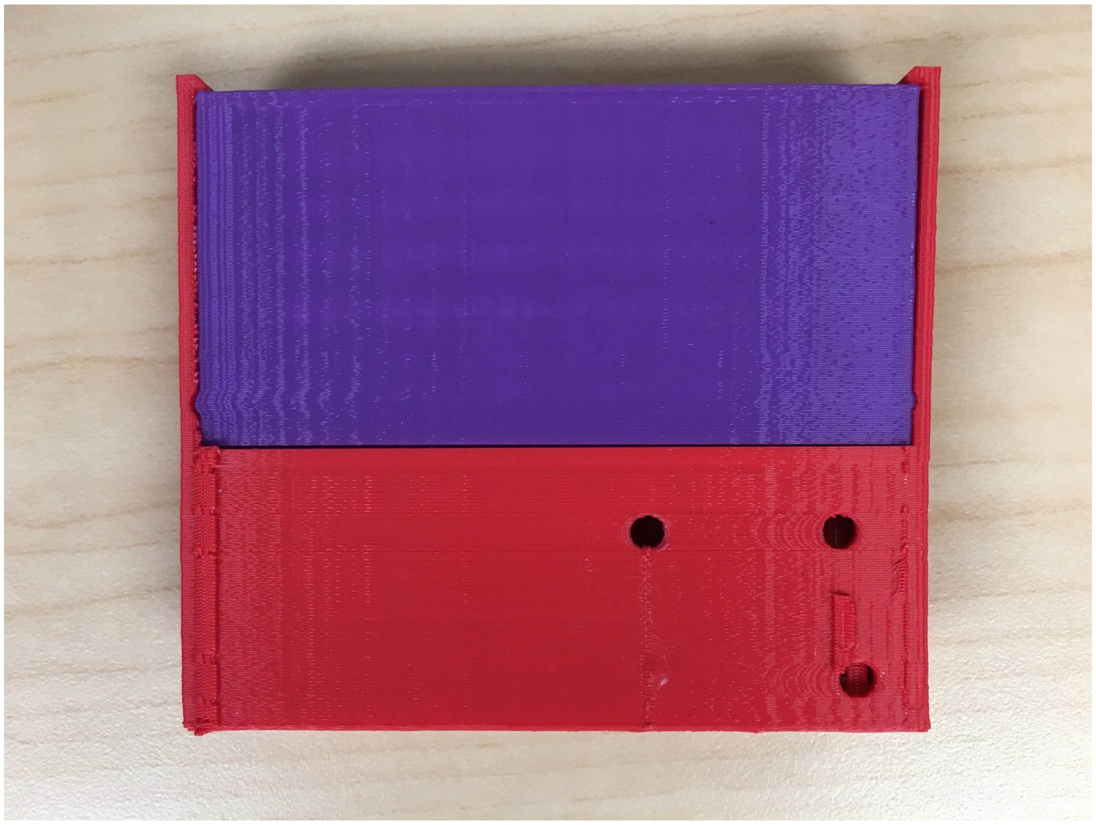

Figure 7. RC Black Box Enclosure

2.4 Architecture

The RC Black Box project has three main software components: data collection, data serving and storage, and the user application. The application workflow is depicted in figure 8. Each component was written in order to be completely modular. The components could be removed and replaced with similar component and the functionality should still work. Each individual component's architecture will be discussed below.

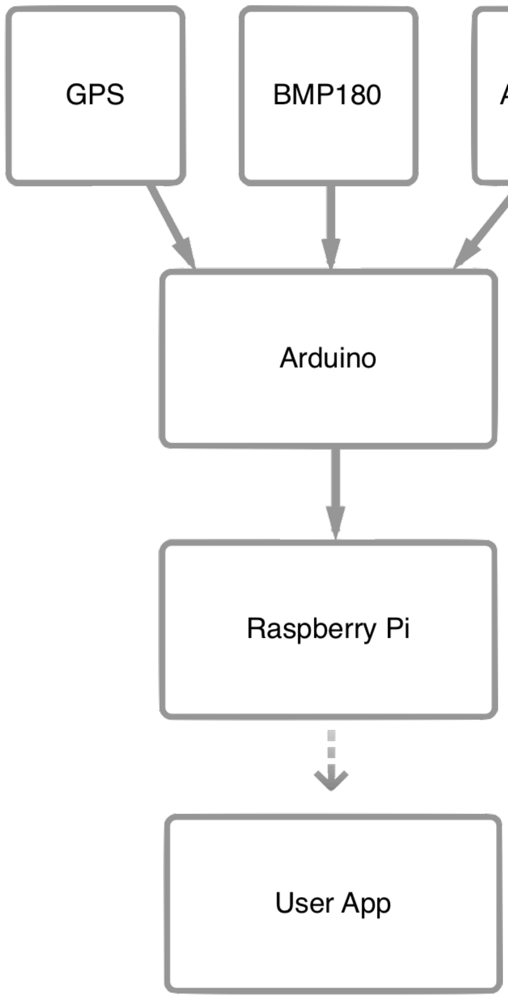

Figure 8. RC Black Box High Level Workflow

2.4.1 Arduino Code Architecture

The *Arduino* code consists of three sensor components, one data collection/ compilation component, and one data sending component as shown in figure 9. The Accelerometer, Barometer, and GPS provide their own data and the *Main* function call collects the data and uses the *SerialCommunication* functions in order to send the data through software serial pins to the *Raspberry Pi.*

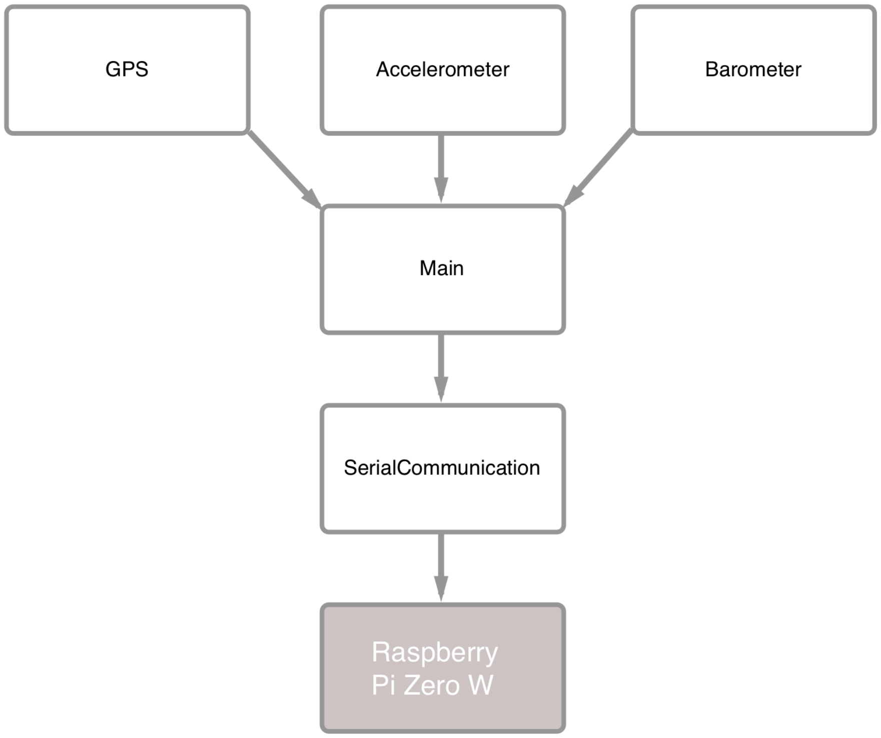

Figure 9. Arduino Code Architecture Diagram

Each device has its own library that allows for data collection. Once the data is collected the data will then be turned into an *Arduino String.* The *Main* file will collect each of these strings and send it off to the *SerialCommunication* file where a checksum will be generated and a final sentence will be constructed. This sentence will then be sent to the *Raspberry Pi*. The code implementation and more detailed analysis is in section 4.1.

2.4.2 Raspberry Pi Code Architecture

The *Raspberry Pi* code consists of several components: the sqlite3 database, the database *Python* file, the server *Python* file, and the parse *Python* file. The figure below details how the architecture is structured. The grey block titled 'Arduino' is the data sent from the *Arduino*. The grey circular arrows denote that the code runs continuously when the device is booted.

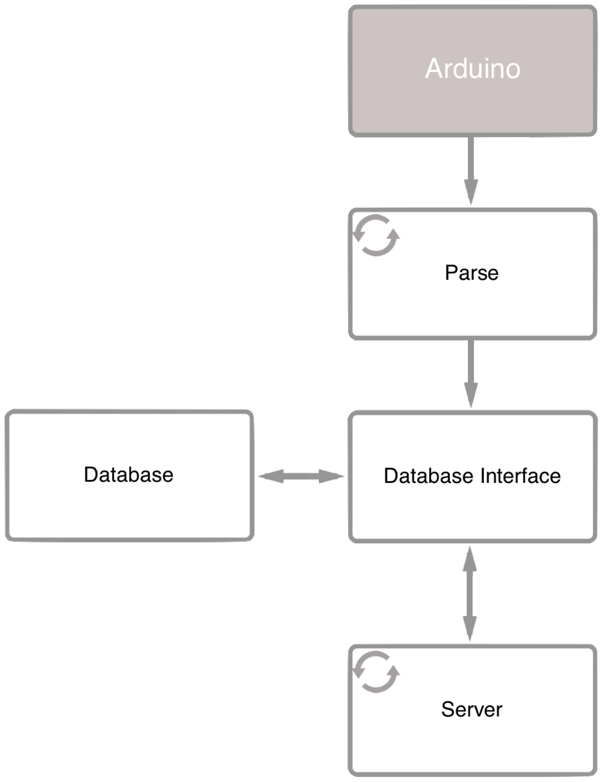

Figure 10. Raspberry Pi Zero Code Architecture Diagram

The sentences retrieved from the *Arduino* are parsed by *parse.py*. They are then used by the database interface in order to construct a *sqlite3* query that inserts the data into the database. *Parse.py* is continuously running alongside the server which is accepting commands from the user application and returning the appropriate data. These processes start at boot and do not terminate until the device is powered off. The code implementation and more detailed analysis is in section 4.2.

2.4.3 User Application Code Architecture

The architecture for the user application consists of three main parts: the socket connection, the data parser, and the views. The workflow of the application is shown below. Solid solid grey lines denote classes that interact with each other. The light grey line connections denote protocols that are associated with a view controller. Each view controller has a view associated with it; they are labelled with a '/'. The view controller that will be displaying the live information to the user is highlighted with light blue text.

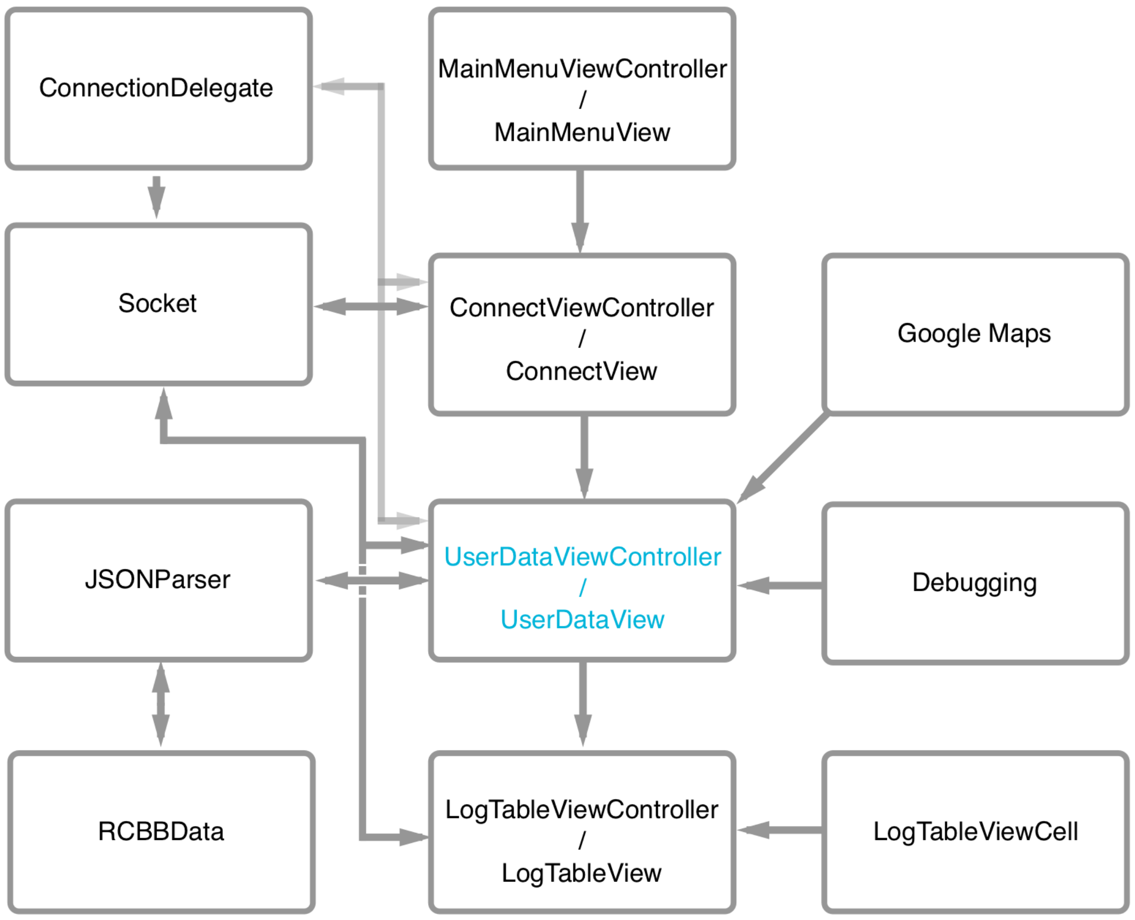
ConnectionDelegate
MainMenuViewController / MainMenuView
Socket
ConnectViewController / ConnectView
Google Maps
JSONParser
UserDataViewController / UserDataView
Debugging
RCBBData
LogTableViewController / LogTableView
LogTableViewCell

Figure 11. User Application Code Architecture Diagram

The code is structured around the *Socket* class which performs the communication with the server. The *UserDataView* is the main view that the user will interact with. It will use the socket in order to send and retrieve data and use the *JSONParse* class and RCBBData struct to display that data. The code implementation and more detailed analysis is in section 4.3.

3.0 Application Implementation

The software for the RC Black Box is comprised of several components that interact with each other closely. All debugging code will be omitted in code examples because it is not necessary in order to understand the code.

3.1 Arduino Code Implementation

The code written for the *Arduino* consists of reading the three sensors and transmitting the data to the *Raspberry Pi.* Below are detailed explanations of each component.

3.1.1 Accelerometer

The accelerometer reading is the quickest and easiest of all three sensors. Readings are taken from pins *A0, A1*, and *A2.* The pins are read using the *Arduino* function *analogRead*. The value read is then mapped to degrees using the *Arduino* libraries *map* function. The *map* function takes the original value, original minimum and maximum values, and two new minimum and maximum values (*Arduino Map Function, 2017*). The output is the original value mapped to a range between the two new minimum and maximum based on the original range. The original minimum and maximum chosen were 290 and 430, respectively. Initially, there was a plan that the device could be calibrated by the user, but, due to time constraints it was decided that the device would use hard coded values based on the ranges read from by the *analogRead* function during initial testing. The mapping function usage was found in a blog written by user 'ilyasaaabdulrahman' (User 'ilyasaaabdulrahman', 2014).

Each axis is read, mapped to degrees, and concatenated into an *Arduino String* object in the form of: "X:[x value],Y:[y value],Z:[z value];". This value will then be used in the final sentence constructed by the *SerialCommunication* library (described in section 4.1.5). Below is the accelerometer code in its entirety:

```
int readAccelX (void) {
    int x = analogRead(xPin);
    x = map(x, minG, maxG, -90, 90);
    return x;
}

int readAccelY (void) {
    int y = analogRead(yPin) - zeroGY;
    y = map(y, minG, maxG, -90, 90);
    return y;
}

int readAccelZ (void) {
    int z = analogRead(zPin) - zeroGZ;
    z = map(z, minG, maxG, -90, 90);
    return z;
}

String getAccelerometerReading (void) {
    String x = String(readAccelX());
    String y = String(readAccelY());
    String z = String(readAccelZ());
    return "X:" + x + ",Y:" + y + ",Z:" + z;
}
```

3.1.2 Barometer

The BMP180 requires significantly more work in order to retrieve data. The device first needs to be calibrated every time the device is powered on. This entails reading the calibration data from the EEPROM of the device (BOSCH, 2013, p. 15). The code for this is shown below:

```
    int calibrateBMP180 () {
        AC1 = readShort(0xAA, 0xAB);
        AC2 = readShort(0xAC, 0xAD);
        AC3 = readShort(0xAE, 0xAF);
        AC4 = readUnsignedShort(0xB0, 0xB1);
        AC5 = readUnsignedShort(0xB2, 0xB3);
        AC6 = readUnsignedShort(0xB4, 0xB5);
        BB1 = readShort(0xB6, 0xB7);
        BB2 = readShort(0xB8, 0xB9);
        MB = readShort(0xBA, 0xBB);
        MC = readShort(0xBC, 0xBD);
        MD = readShort(0xBE, 0xBF);
        return 0;
    }

    short readShort (char MSBaddress, char LSBaddress) {
        short MSB;
        short LSB;
        MSB = (short)readRegister(MSBaddress) << 8;
        LSB = (short)readRegister(LSBaddress);
        return MSB | LSB;
    }

    unsigned char readRegister (char address) {
        int error;
        unsigned char value;
        Wire.setClock(100000);
        Wire.beginTransmission(BMP180Address);
        Wire.write(address);
        error = Wire.endTransmission();
        if (error == 0) {
            Wire.requestFrom(BMP180Address, 1);
            value = Wire.read();
        }
        return value;
    }
```

The *calibrateBMP180* function reads the significant bits from the EEPROM in the calibration variables. It utilises the function *readShort* which reads the registers provided using the *readRegister* function. The *readRegister* function uses the *Arduino Wire* library to try to read the register and return the value read from it (*Arduino Wire Library,* 2017). The values in the *calibrateBMP180* function will later be used to calculate the true temperature and pressure. The reason this needs to be done is because the values for each BMP180 device will be different and this cannot be hard coded (BOSCH, 2013, p. 13). The calibration coefficients as provided by BOSCH 2013, are provided in the table below:

Value	MSB	LSB
AC1	0xAA	0xAB
AC2	0xAC	0xAD
AC3	0xAE	0xAF
AC4	0xB0	0xB1
AC5	0xB2	0xB3
AC6	0xB4	0xB5
B1	0xB6	0xB7
B2	0xB8	0xB9
MB	0xBA	0xBB
MC	0xBC	0xBD
MD	0xBE	0xBF

Table 1. BMP180 Calibration Values

Once the device is calibrated, the device can be read. In order to read data a timer is used. To get the raw temperature the timer is started, the *TemperatureMeasureRegister* value (0x2E) is written into the *ControlMeasureRegister* (0xF4). Once five seconds have elapsed then the raw temperature value is received by reading the registers 0xF6 and 0xF7 (BOSCH, 2013, p. 15). The function is provided below:

```
int startTemperatureMeasurement () {
    int error;
    error = writeToRegister(ControlMeasureRegister,
TemperatureMeasureRegister);
    return error;
}

long getUncompensatedTemperature () {
    long MSB, LSB;
    MSB = readRegister(MSBRegister);
    LSB = readRegister(LSBRegister);
    return (long)((MSB << 8) + LSB);
}
```

Once the raw temperature value is read the *PressureMeasureRegister* value (0x34) is read into the *ControlMeasureRegister* (0xF4). Once a certain amount of time has elapsed — this depends on the oversampling setting, in the project it is set at 0 for ultra-low power mode, meaning the delay would be 5 seconds — then the raw pressure value can be retrieved. This is done similarly to the temperature reading, except that we read registers 0xF6, 0xF7, and 0xF8 (BOSCH, 2013, p. 15). The function is shown below:

```
int startPressureMeasurement () {
    int error;
    error = writeToRegister(ControlMeasureRegister,
PressureMeasureRegister + (overSamplingSetting << 6));
    return error;
}

long getUncompensatedPressure () {
    long MSB, LSB, XLSB;
    MSB = readRegister(MSBRegister);
    LSB = readRegister(LSBRegister);
    XLSB = readRegister(XLSBRegister);
    return long(((MSB << 16) + (LSB << 8) + XLSB) >> (8 -
overSamplingSetting));
}
```

As soon as the raw values are read, the timer is restarted and several calculations are done in order to produce the real temperature and pressure. These calculations are all listed in the BMP180 datasheet and are not in the scope of this document (BOSCH, 2013, p. 15).

3.1.3 GPS

Communication with the GPS module is done using the *Arduino HardwareSerial* library. This is done by creating a new *HardwareSerial* object in the *Main* file and printing to and reading from it. There are two steps for reading data from the GPS module. First, initialisation is done at every boot. The *initialiseGPS* method writes three sentences to the *HardwareSerial* object: what kind of data it wants, the update rate, and the antenna status. In the case of the RC Black Box, we need to retrieve all the data possible, we need an update rate of one second, and we have no antenna (GlobalTop Tech Inc, 2012, p. 8, 12). The function is shown below:

```
int initialiseGPS (HardwareSerial *serial) {
    serial->begin(9600);
    serial->println(nmeaOutputRMCGGA);
    serial->println(nmeaUpdate);
    serial->println(antennaStatus);
    delay(1000);
    serial->println("$PMTK605*31");
    return 0;
}
```

Once initialisation is done, we can start reading data from the device. This is done by calling the *getGPSData* function in the *Main* loop. Each time the function is called a character is appended to the running NMEA sentence — an NMEA sentence is the string value returned by the GPS which contains all of the information about the current location and other values. Once a *newline* is detected, the sentence is complete and it can be used. *getGPSData* is shown below:

```
boolean getGPSData (HardwareSerial *serial, String &nmea) {
    char c;
    if (!serial->available()) return false;
    c = (char)serial->read();
    if (c == '\n') {
        c = 0;
        nmea += c;
        return true;
    }
    nmea += c;
    return false;
}
```

Note: The *Adafruit* website provided a basic reading and writing test that was used to implement the code above (User 'lady ada', 2017); it was used in conjunction with the Adafruit Ultimate GPS Featherwing datasheet to get the correct results (GlobalTop Tech Inc, 2011).

3.1.4 Main

The *Main Arduino* file is where all of the data collection code is utilised. The timer and interrupt for the BMP180 are defined here. In the *setup* function all of the devices are initialised and the serial communication is started at a Baud rate of 9600. The *loop* function is executed continuously while the device is powered on and retrieves all of the readings using the methods described above. Once a complete NMEA sentence is retrieved from the GPS module the *loop* function takes all of the data that has been read and sends it to the *Raspberry Pi* using the functions defined in the *SerialCommunications* file.

3.1.5 SerialCommunication

The *SerialCommunication* file has two responsibilities: writing compiled sentences to the *Raspberry Pi* using the *Arduino SoftwareSerial* library and creating a checksum for that sentence. The process is as follows: the *loop* function in *Main* calls *writeToSerial* with the *SoftwareSerial* object and three strings representing the data from the three sensors. Then a checksum is created for the data provided. Finally, the data is compiled into a sentence and sent to the *Pi* using the *SoftwareSerial* object.

A note on the *SoftwareSerial* object: the GPS module operates using the only serial pins on the *Arduino Nano.* Therefore it was necessary to create 'virtual' serial pins using pins D6 and D7 in order to interface with the *Pi* (*Arduino SoftwareSerial Library, 2017*).

3.2 Raspberry Pi Code Implementation

The software written for the *Pi* consists of receiving data from the *Arduino*, updating the database with that data, and sending the data to the user application. The separate components are detailed below.

3.2.1 Parse

The *Arduino* sends data to the *Pi* in the form of sentences, detailed as:

```
@#75ADXL:X:-1,Y:-18,Z:-55;BMP180:T24.80,P93.32;GPS:$GPRMC,
220516,A,53.539848,N,113.480256,W,173.8,231.8,130694,004.2,W*7D
```

The data is read with the help of the *Python pySerial* library (Liechti, 2017). The sentences begin with an '@' symbol denoting that a new sentence is starting, followed by a '#' symbol and a checksum value that is used to determine if the sentence provided is fragmented or not. Following the head of the sentence is the data collected by the *Arduino*. The *parse* module (shown in figure 10) is responsible for parsing the sentences provided, constructing a *sqlite3* query, and having the database interface module execute the query against the *sqlite3* database. The code below details the main method call that is run continuously when the device is powered on:

```python
def main():
    uart = serial.Serial('/dev/serial0',9600,timeout=1)
    try:
        while 1:
            available = uart.inWaiting()
            if available > 0:
                latest = uart.readline()
                uart.flushInput()
                if latest.startswith("@") and latest != "":
                    if (correctHash(latest, True)):
                        parseAndUpdate(latest)
                        sleep(1)
    except KeyboardInterrupt:
        uart.close()
```

After a line is read and the line starts with an ‘@‘ symbol the checksum is verified. If the checksum is correct, then the sentence is parsed and the query constructed is executed against the database.

3.2.2 Database & Database Interface

The database is a *sqlite3* database that consists of one table: data. The data table holds the information collected by the *Arduino*. The timestamp is the primary/unique key that uses millisecond precision — this allows the device to collect multiple data per second. The database interface consists of two functions: *executeQuery* and *executeNonQuery*. These functions are almost identical except that *executeQuery* returns values from the database, while *executeNonQuery* only alters the database. This file utilises the *Python* library *sqlite3*, which allows for executing queries against a *sqlite3* database (Python Software Foundation, 2017). The two database functions are shown below:

```
def executeNonQuery (query):
    connection = sqlite3.connect(path)
    cursor = connection.cursor()
    cursor.execute(query)
    connection.commit()
    connection.close()

def executeQuery (query):
    connection = sqlite3.connect(path)
    cursor = connection.cursor()
    cursor.execute(query)
    data = cursor.fetchall()
    connection.commit()
    connection.close()
    return data
```

3.2.3 Server

The server *Python* file runs continuously alongside the parsing file. The server utilises the wireless access point that is emitted by the *Raspberry Pi*. The server starts listening in the *init* method call of the *Server* class seen below:

```
def __init__ (self):
    self.mainSocket = socket.socket()
    self.host = socket.gethostbyname(check_output(["hostname", "-I"]))
    self.port = 3000
    self.setupMessage = "Server = Host: " + self.host + ", Port: " +
str(self.port)
    print(self.setupMessage)
    self.mainSocket.setsockopt(socket.SOL_SOCKET, socket.SO_REUSEADDR,
1)
    self.mainSocket.bind((self.host,self.port))
    self.mainSocket.listen(5)
    self.connection, self.address = self.mainSocket.accept()
    self.endMessage = "Connection ended."
```

Once the server is started, it immediately starts trying to receive information from the user application. The user application can send one of five commands: ‘L’ to request the latest entry in the database, ’T’ for all the entries that were created that day, ‘A’ to retrieve all the entries in the database, and ‘Q’ or ‘E’ in order to disconnect. The two methods responsible for performing these actions are shown below:

```
def receiveBytes (self):
    while True:
        data =
self.connection.recv(1024).replace("\r\n",'').upper()
        if data == "Q" or data == "E" or data =="":
            break
            connection.send(self.endMessage)
            print(self.endMessage)
            connection.close()
        else:
            print(data)
            self.dispatch(data)

def dispatch (self, data):
    if data == "L" or data == "A" or data == "T":
        query = "SELECT * FROM data ORDER BY date DESC LIMIT 1"
        if data == "A":
            query = "SELECT * FROM data ORDER BY date"
        if data == "T":
            query = "SELECT * FROM data WHERE date >=
date('now')"
        results = database.executeQuery(query)
        self.sendResults(results)
```

Once a command is received by the server from the *iOS* application, a *sqlite3* query is constructed and executed against the database. The results from the query are then sent to the *sendResults* method which constructs a JSON string (example provided below) that is then sent to the user application.

```
{
    "date":"03/04/2017 20:03:03",
    "x" : "1.23",
    "y" : "2.34",
    "z" : "3.45",
    "temperature" : "23.45",
    "pressure" : "93.55",
    "altitude" : "105",
    "latitude" : "53.5678694",
    "latDir" : "N",
    "longitude" : "-113.5045351",
    "longDir" : "W",
    "quality" : "0",
    "satellites" : "04",
    "speed" : "40"
}
```

3.3 User Application Code Implementation

The user application is written with the MVC (model-view-controller) software pattern in mind — “the Controller creates Views and coordinates Views and Models” (Reenskaug & Coplien, 2009). Each component of the application is detailed below.

3.3.1 MainMenuView

Initially, the user is greeted at the main menu. Tapping the ‘Start’ button will allow the user to begin connecting to the device. There is an optional debug menu item that is used for debugging the application, this would not be present for a final release of the application. Tapping the ‘Start’ button moves to the *ConnectView* from the *MainMenuView.*

3.3.2 ConnectView

The socket connection is started when the user taps on the ‘Connect’ button (the network connection must have already been established of course). The socket is initialised by passing the host address, the port, and the view controller as the delegate. Finally, we can connect to the host by starting the stream, designating the input and output stream delegates, and opening the streams as shown below:

```
init (withHost h: String, withPort p: Int, withDelegate
delegate: ConnectionDelegate) {
        _host = h
        _port = p
        self.delegate = delegate
    }

func ConnectToHost () -> Int {
        if _inputStream != nil || _outputStream != nil {
            DisconnectFromHost()
            _inputStream = nil
            _outputStream = nil
        }
        Stream.getStreamsToHost(withName: _host, port: _port,
inputStream: &_inputStream, outputStream: &_outputStream)
        if _inputStream != nil && _outputStream != nil {
            _inputStream?.delegate = self
            _outputStream?.delegate = self
            _inputStream?.schedule(in: .main,
forMode: .defaultRunLoopMode)
            _outputStream?.schedule(in: .main,
forMode: .defaultRunLoopMode)
            _inputStream!.open()
            _outputStream!.open()
            return 0
        }
        return 1
    }
```

3.3.3 UserDataView

Once connected, the application moves from the *ConnectView* to the *UserDataView* and starts sending the default command, ‘L’, to the server. The ‘L’ command will result in a JSON string that contains the latest datum from the database stored on the *Pi*. The JSON string is then decoded and the *UserDataView* displays the data to the user. This will repeat until the user either exits the application or chooses one of the other commands (by tapping on the corresponding button).

3.3.4 LogTableView

While in the *UserDataView*, the user can choose to either view all of the data available in the database, or just the data from that day. Tapping on either of the buttons will move the current view to the *LogTableView* and send either the ’T’ or ‘A’ command to the server. The server will then respond accordingly and the *LogTableViewController* will parse the JSON data into RCBBData structs and construct *LogTableViewCell*’s to display. Once the data has been prepared, the user can then view the data in a table or tap the back button to go back to the *UserDataView*.

3.3.5 Socket & ConnectionDelegate

The *Socket* class is the component of the application that is doing the most work. The initialisation of the socket is described and shown in section 3.4.3.2. The other two main methods of the *Socket* class are *WriteBytes and ReadBytes. WriteBytes* will send the given string to the server; the given string is always one of the aforementioned commands ‘L’, ’T’, ‘A’, ‘Q’, or ‘E’. The method is shown below:

```
func WriteBytes (data: String) {
        let data = data.data(using: String.Encoding.utf8, 
allowLossyConversion: false)
        data?.withUnsafeBytes { _outputStream?.write($0, 
maxLength: (data?.count)!) }
    }
```

On the other hand, *ReceiveBytes* checks if there are any bytes ready to be read and loads them into the buffer. It then copies the information of the buffer to another buffer in order to retain the data and clears the original buffer. This prevents data fragmentation. *ReceiveBytes* is shown below:

```
func ReadBytes () -> String {
        var bytes = [UInt8]()
        while (_inputStream?.hasBytesAvailable)! {
            _buffer = [UInt8](repeating: 0, count: 512)
            _inputStream!.read(&_buffer, maxLength: 
_buffer.count)
            RetainInformation(from: _buffer, to: &bytes)
        }
        return String(bytes: bytes, encoding: 
String.Encoding.utf8)!
    }
```

The Socket class also accepts a delegate in order to alert the application of any stream errors. This is done by implementing the following code:

```
func stream(_ stream: Stream, handle eventCode: Stream.Event) {
        switch eventCode {
        case Stream.Event.errorOccurred:
            socketError = stream.streamError.debugDescription
            Connected(message: "Retry")
        case Stream.Event.endEncountered:
            socketError = "Disconnected"
        default:
            socketError = nil
            Connected(message: "Connected")
        }
    }
```

```
func Connected (message: String) {
        delegate?.ConnectedStatus(message: message)
    }
```

The *stream* method is an optional callback for the input/output streams; if any events occur within those streams, this method will be called and it will call other methods accordingly. This optional callback allows us to use the *Connected* method from *ConnectionDelegate* protocol — this allows us to alert the user to the status of the stream connection. This would be done for disconnects or unsuccessful connection attempts.

3.3.6 JSONParser && RCBBData

These two data structures are presented together because they work in close conjunction with one another and do not need separate explanations. The JSONParser class provides a single static method that accepts a string and deserialises it directly into a RCBBData struct directly by using the RCBBData initialiser. The code for deserialising the JSON object (Apple, 2017) is depicted below:

```
static func Decode (data: String) -> RCBBData? {
        let formattedData = data.components(separatedBy: "\n")[0]
        if formattedData.lowercased().range(of: "\0") == nil {
            let jsonData = formattedData.data(using:
String.Encoding.utf8, allowLossyConversion: false)!
            do {
                let json = try JSONSerialization.jsonObject(with:
jsonData, options: []) as! [String: String]
                if let RCBBData = RCBBData(json: json) {
                    return RCBBData
                }
            } catch {
                print("Failed to parse.")
            }
        }
        return nil
    }
```

The initialiser accepts a JSON object in the form of a key/value pair and assigns the fields accordingly (Apple, 2017).

```
 init? (json: [String: Any]) {
        self.date = json["date"] as! String
        self.x = json["x"] as! String
        self.y = json["y"] as! String
        self.z = json["z"] as! String
        self.temperature = json["temperature"] as! String
        self.pressure = json["pressure"] as! String
        self.altitude = json["altitude"] as! String
        self.latitude = json["latitude"] as! String
        self.latitudeDirection = json["latDir"] as! String
        self.longitude = json["longitude"] as! String
        self.longitudeDirection = json["longDir"] as! String
        self.quality = json["quality"] as! String
        self.satellites = json["satellites"] as! String
        self.speed = json["speed"] as! String
    }
```

3.3.7 Google Maps & Debugging

The debugging segment consists of three view controllers: one that displays the instructions to the person debugging, another that provides the controls for debugging, and the last that provides a way to view the debugging data. This view will not be covered in more depth because it was used strictly for testing and in the final product the user will not be able to access this section of the application.

In order to display the location of the device using the collected geographical coordinates, the *Google Maps* API was used. In order to use the API, several steps had to be followed. Firstly, the project needed to install the dependencies using *Cocoapods*. *Cocoapods* is a dependency manager that is *macOS* specific. Once installed, an API key had to be retrieved — seeing as this is for a non-profit application there is no license fee for acquiring this key. Once the dependency is installed and the key referenced in the *AppDelegate* file, the API can be used (Google, 2017).

To display the map the following code was written (Google, 2017):

```
private func UpdateMapView (with datum: RCBBData) {
        if let coordinates = ConvertCoordinates(latitude:
datum.latitude, latDir: datum.latitudeDirection, longitude:
datum.longitude, longDir: datum.longitudeDirection) {
            if IsBeyond(coordinates, and: latestCoordinates)  {
                latestCoordinates = coordinates
                mapView.clear()
                let camera = GMSCameraPosition.camera(withLatitude:
coordinates[0], longitude: coordinates[1], zoom: 14)
                let marker = GMSMarker(position:
CLLocationCoordinate2D(latitude: coordinates[0], longitude:
coordinates[1]))
                marker.map = mapView
                mapView.camera = camera
            }
        }
    }
```

First, the coordinates are converted to real coordinates using the directional symbol provided by the server (i.e. N, S, E, W) and then the coordinates are checked to determine if they are significantly different from the last coordinates of the device. If they are then we can change the camera and move the marker.

4.0 Project Results

The project has met all of the basic requirements that were outlined at the beginning of this report, although, there can still be several improvements in almost all areas. The physical build is a mess of wires and subpar soldering and it is still bigger than was initially desired. There is no ability to view each data log individually and the general user interface needs significant work. Several features were omitted due to time constraints, such as a printed circuit board, accelerometer calibration, and wiring a small antenna to the GPS module for a better signal.

Also, the price is more than a device like this should cost. The price for the individual components cost more than $150 and a few of the parts are extremely hard to find (i.e. the *Raspberry Pi Zero W*). This alone would deter and consumer from purchasing this device, but, it also is not as feature-rich as it could be.

However, this device has turned out to be fairly sophisticated. It allows for live data retrieval from an RC vehicle as intended, it is physically small and light, and the software is written in a way that it can be easily expanded.

5.0 Conclusion

The RC Black Box is a device that can provide the user with valuable metrics about their RC vehicle live on their phone. It uses three sensors: an accelerometer, a barometer, and a GPS module that work together with an *Arduino Nano* and a *Raspberry Pi Zero W*. And all of this capability is housed within a small 7.5cm x 8.5cm enclosure.

This report examined the several hardware and software components of the RC Black Box. It detailed the hardware connections between the three sensors, the *Arduino*, and the *Raspberry Pi.* And it went through the specifics of how the devices communicate with each other on the software side. The report also went through a brief overview of how each sensor worked and communicated its data.

The RC Black Box uses several pieces of robust software that allow the device to read and send data to the user in half second intervals. The device would be an asset for RC vehicle hobbyists because of the fact that is it relatively small and light, but, also because the application is easy to use and provides valuable metrics.

In summary, the RC Black Box is an interesting piece of hardware and software. The development of this product is quite complicated and there is room for improvement in many areas. The device meets the basic requirements specified before the project started, however, a consumer would not be impressed by the lack of features and the high price.

6.0 Reflection

This project has taken a lot out of me. I have had to use almost all of the knowledge learned during my time in CNT and I had to do a significant amount of outside studying in order to finish this project. The project has several components that all have to interact with each other and keeping track of all the moving parts was a small project on its own. Even with all of my effort the project still did not come out as I had initially planned (i.e. it is not as polished as I would like, the size is still too large, etc.).

When I found out that I was going to be finishing the project alone I was excited. I generally work better alone anyway. My partner leaving allowed me to structure the hardware and software exactly how I thought was best, but, it also limited the project. Because I did not have an outside influence I became stuck in my ways and I feel the project suffered. There are aspects of the project that could have been done more efficiently, but, were done the way that that first came to thought. And once implemented there was not enough time to go back and reconsider many aspects.

While I do feel that the project may have been stifled due to the lack of a partner, I feel that I learned more than I would have otherwise. I gained a better understanding of C/C++, Python, Swift, serial communication, networking, databases, and I learned how to solder (thought, I am still quite bad at it). The project may not have turned out as perfectly as I thought it would, but, I am satisfied with the results and even more satisfied that I was able to push through and learned a large variety of different skills.

References

ADXL335 Datasheet. (2009). Analog Devices Inc. Retrieved January 13, 2017, from http://www.analog.com/media/en/technical-documentation/data-sheets/ADXL335.pdf

Apple. (2017, September 12). Working with JSON in Swift. Retrieved March 25, 2017, from https://developer.apple.com/swift/blog/?id=37

Arduino Map Function. (2017). Retrieved March 15, 2017, from *https://www.arduino.cc/en/reference/map*

Arduino Wire Library. (2017). Retrieved Februrary 13, 2017, from *https://www.arduino.cc/en/reference/wire*

Arduino SoftwareSerial Library. (2017). Retrieved February 16, 2017, from *https://www.arduino.cc/en/Reference/softwareSerial*

Arduino Serial Library. (2017). Retrieved March 10, 2017, from *https://www.arduino.cc/en/reference/serial*

BOSCH. (2013, April 5). BMP180 Digital Pressure Sensor Data Sheet. Retrieved January 20, 2017, from https://cdn-shop.adafruit.com/datasheets/BST-BMP180-DS000-09.pdf

Chris Liechti. (2017). *pySerial API*. Retrieved March 14, 2017, from *http://pyserial.readthedocs.io/en/latest/pyserial_api.html*

GlobalTop Tech Inc. (2011). FGPMMOPA6H GPS Standalone Data Sheet. Retrieved February 5, 2017, from https://cdn-shop.adafruit.com/datasheets/GlobalTop-FGPMMOPA6H-Datasheet-V0A.pdf

GlobalTop Tech Inc. (2012). PMTK Command Packet [PDF]. Retrieved February 5, 2017, from https://cdn-shop.adafruit.com/datasheets/PMTK_A11.pdf

Google. (2017). *Google Maps SKD for iOS*. Retrieved March 28, 2017, from *https://developers.google.com/maps/documentation/ios-sdk/reference/*

Python Software Foundation. (2017). *DB-API 2.0 interface for SQLite databases*. Retrieved March 20, 2017, from *https://docs.python.org/2/library/sqlite3.html*

Reenskaug, T., & Coplien, J. O. (2009, March 20). Objects are principally about people and their mental models—not polymorphism, coupling and cohesion. Retrieved March 10, 2017, from http://www.artima.com/articles/dci_vision.html

User ‘ilyasaaabdulrahman’. (2014, Sept. 14). Accelerometer ADXL335 Fast Test and Angle Test. Retrieved February 3, 2017, from http://tutorial.cytron.com.my/2014/09/15/accelerometer-adxl335-fast-test-and-angle-test/

User ‘lady ada’*.* (2017). *Adafruit Ultimate GPS Featherwing Overview*. Retrieved January 15, 2017, from *https://learn.adafruit.com/adafruit-ultimate-gps-featherwing?view=all*