

Abstract

The aim of this project was to design and build an accurate portable weather station that is capable of monitoring and recording localised weather conditions in South Africa, for a 12-hour duration, using a suitable mobile user interface that communicates over Bluetooth. A Particle Photon was used to evaluate sensor data and transmit the corresponding weather measurement data, over Bluetooth Low Energy, to a smart phone application created on the Blynk platform. The weather measurements were produced by sensors with low power requirements: the Bosch BME280 (temperature, pressure and relative humidity), Davis 6410 (wind speed and direction) and the BGT WS-601ABS2 tipping bucket (rain). These sensors were tested against acceptable standards and the level of accuracy required from each sensor, set out by the design specifications, was achieved. The weather station application was designed, not only to be functional but, to be aesthetically pleasing and user friendly. This portable weather monitoring system was found to reliably log weather data and, using a rechargeable 3.7 V 1050 mAh lithium ion battery, was tested to be operational for a minimum of 16 hours and 41 minutes.

TABLE OF CONTENTS

PRELIMINARIES

Acknowledgements.....	ii
Abstract.....	iii
Table of Contents	iv
List of Figures.....	ix
List of Tables.....	xi
Constants and Abbreviations	xii

CHAPTER 1: INTRODUCTION

1.1	Design Specifications	1
1.2	Project Process.....	3
1.3	Project Limitations	3

CHAPTER 2: DESIGN

2.1	The Block Diagram	5
2.1.1	The Photon	6
2.1.2	Temperature, Pressure and Humidity Sensor	6
2.1.3	Wind Speed and Direction Sensor	7
2.1.4	Rain Sensor	7
2.1.5	Bluetooth Module	8
2.1.6	Data Logging using a Micro SD Card Module	9
2.1.7	Power Supply Circuitry: Rechargeable Battery	9
2.1.8	Power Supply Circuitry: Charge Controller.....	10
2.1.9	Battery Status Monitoring Feature	11
2.1.10	Charge Indication LED	12
2.1.11	Connector and Cable Selection	12
2.1.12	Mobile App	12

CHAPTER 3: BUILD

3.1	The Spare I2C Port	14
3.2	The BME280 Temperature, Humidity and Pressure Sensor	15
3.3	Wind Sensor	15
3.3.1	Repairs	15
3.3.2	Wiring and Circuit Diagram	16
3.4	Rain Sensor	17
3.5	Bluetooth	18
3.6	Micro SD Card Module.....	19
3.7	Power Supply Circuitry.....	19
3.8	Battery Status Monitoring	20
3.9	Charge Indication Circuit	20
3.10	Design of the PCB.....	21
3.11	The Mobile Application.....	23
3.12	Enclosure Selection and Modification	25

CHAPTER 4: FIRMWARE DESIGN

4.1	The “void loop()” Subroutine.....	28
4.2	The “AllSubroutines()” Subroutine	28
4.3	BME280 Operation and Programming	29
4.4	Davis Wind Sensor Operation and Programming.....	30
4.5	Rain Sensor Operation and Programming	34
4.6	Power Saving Features.....	36
4.7	Mobile App Operation and Programming	37
4.7.1	The Blynk Security Feature	38
4.8	The Operation and Programming of the Battery Status Monitoring Feature	38
4.9	Data Logging Operation and Programming	42
4.10	Problems Encountered and their Solutions.....	45
4.10.1	Obtaining and Maintaining Time on the Photon.....	45
4.10.2	Fault Finding and Debugging.....	46

CHAPTER 5: HARDWARE TESTING AND RESULTS

5.1	Davis Wind Speed Sensor	48
-----	-------------------------------	----

5.1.1	Wind Speed Sensor Testing	48
5.1.2	Wind Speed Sensor Test Results	48
5.2	Davis Wind Direction Sensor	51
5.2.1	Wind Direction Sensor Testing	51
5.2.2	Wind Direction Sensor Test Results	53
5.3	BME280 Temperature Sensor.....	54
5.3.1	Temperature Sensor Testing	54
5.3.2	Temperature Sensor Test Results.....	54
5.4	BME280 Humidity Sensor	56
5.4.1	Humidity Sensor Testing.....	56
5.4.2	Humidity Sensor Test Results	56
5.5	BME280 Pressure Sensor	59
5.5.1	Pressure Sensor Testing	59
5.5.2	Pressure Sensor Test Results.....	59
5.6	BGT WS-601ABS2 Rain Sensor	62
5.6.1	Rain Sensor Testing	62
5.6.2	Rain Sensor Test Results.....	62
5.7	Sensor Connection Testing and Results.....	64
5.8	Power Consumption	65
5.8.1	Current Consumption Testing	65
5.8.2	Current Consumption Test Results.....	65
5.9	Data Logger	66
5.9.1	Data Logger Testing.....	66
5.9.2	Data Logger Test Results	68
5.10	Timestamp	71
5.10.1	Timestamp Testing	71
5.10.2	Timestamp Test Results.....	74
5.11	Photon's Voltage Measurement.....	78
5.11.1	Voltage Measurement Testing	78
5.11.2	Voltage Measurement Test Results	79
5.12	Battery Status Monitoring	80
5.12.1	Battery Status Testing	80
5.12.2	Battery Status Test Results.....	83
5.13	Firmware Execution Period.....	84
5.13.1	Firmware Execution Period Testing.....	84

5.13.2	Firmware Execution Period Test Results	85
5.14	Period of Operation	85
5.14.1	Period of Operation Testing	86
5.14.2	Period of Operation Test Results.....	87
5.15	Problems Encountered and their Solutions.....	87
5.15.1	Wind Speed Sensor Testing.....	87

CHAPTER 6: MOBILE APPLICATION TESTING AND RESULTS

6.1	Weather Sensor Measurements and Connection Status	89
6.1.1	Testing	89
6.1.2	Results	89
6.2	Battery Status Monitoring	89
6.2.1	Testing	89
6.2.2	Results	94
6.3	Data Logger	101
6.3.1	Testing	101
6.3.2	Results	104
6.4	Communication Range Between the Weather Station Hardware and the App.....	105
6.4.1	Testing	105
6.4.2	Results	106
6.5	Problems Encountered and their Solutions.....	106
6.5.1	The Operating Battery Voltage Range.....	106
6.5.2	Weather Station Hardware and App Connection Interruption.....	111

CHAPTER 7: CONCLUSION

7.1	Conclusion	113
7.2	Recommendations.....	113

REFERENCES	115
-------------------------	------------

Annexure A: Time Management Schedule	121
---	------------

Annexure B: Complete Circuit Schematic	122
---	------------

Annexure C: Enclosure Modifications.....123

Annexure D: Full Code Listing124

LIST OF FIGURES

Figure 2.1:	Block Diagram of the System.....	5
Figure 2.2:	Pinout of the Particle Photon	6
Figure 3.1:	Connection of the Spare I2C Port.....	14
Figure 3.2:	Connection of the BME280 to the Photon	15
Figure 3.3:	Connection of the Davis 6410 Wind Sensor to the Photon.....	16
Figure 3.4:	Connection of the BGT WS-601ABS2 Rain Sensor to the Photon	17
Figure 3.5:	Tipping Bucket Mechanism	18
Figure 3.6:	Connection of the Bluetooth Module to the Photon	19
Figure 3.7:	Connection of the Micro SD Card Module to the Photon	19
Figure 3.8:	Charging Circuit	20
Figure 3.9:	Circuits used to Achieve the Battery Status Indication Feature.....	20
Figure 3.10:	Charge Indication circuit.....	21
Figure 3.11:	Weather Station PCBs	23
Figure 3.12:	Weather Station Application	25
Figure 3.13:	Spacers used to Secure Battery in Enclosure	26
Figure 4.1:	Image Showing a Sample of the Logged Weather Data on the SD Card	45
Figure 5.1:	Adopted Method for Wind Speed Sensor Testing	48
Figure 5.2:	Scatter Graph Showing the Correlation of Wind Speed Measurements between the Davis sensor and the RS212-578 Wind Speed Meter.....	50
Figure 5.3:	Comparison between the Wind Speeds Measured by the Davis Wind Speed Sensor and RS212-578 Wind Speed Meter.....	50
Figure 5.4:	Template Used to Set the Wind Direction Sensor.....	51
Figure 5.5:	Setting of the Wind Direction Sensor Towards North.....	53
Figure 5.6:	Scatter Graph Showing the Correlation of Temperature Measurements between the BME280 Temperature Sensor and the Fluke 52 Thermometer	55

Figure 5.7:	Comparison Between the Temperature Measurements Produced by the BME280 Sensor and the Fluke 52 Thermometer	56
Figure 5.8:	Scatter Graph Showing the Correlation of Measurements Produced by the BME280 Humidity Sensor and the Weather Station Standard.....	58
Figure 5.9:	Comparison Between the Humidity Measurements Produced by the BME280 Sensor and the Weather Station Standard.....	58
Figure 5.10:	Scatter Graph Showing the Correlation of Measurements Produced by the BME280 Pressure Sensor and the Calculated Standard	61
Figure 5.11:	Comparison Between the Humidity Measurements Produced by the BME280 Sensor and the Weather Station Standard.....	61
Figure 5.12:	Scatter Graph Showing the Correlation of Measurements Produced by the BGT WS-601ABS2 Rain Sensor and the Calculated Standard	63
Figure 5.13:	Comparison Between the Rain Measurements Produced by the Rain Sensor and the Calculated Standard	64
Figure 5.14:	The Current Consumption of the Weather Station Before and After Implementing Power Saving Features	66
Figure 5.15:	Test Circuit used to Verify the Accuracy of the Voltage Measurements Produced by the Photon	78
Figure 5.16:	Comparison Between the Voltage Measurements Produced by the Photon and the Standard	79
Figure 5.17:	Comparison Between the Calibrated Voltage Measurements Produced by the Photon's Channel 0 against the Fluke 177 Multimeter Standard	80
Figure 5.18:	Components and Configuration used to Test the Battery Monitoring Feature	81
Figure 5.19:	Graph Showing the Required and Achieved Periods of Operation	87
Figure 5.20:	Initial Method for Wind Speed Sensor Testing	88
Figure 6.1:	Circuit Used to Test the Charging Period of the Battery While it is Disconnected from the Load	90
Figure 6.2:	Testing the Minimum Operating Voltage of the Weather Station.....	107

LIST OF TABLES

Table 1.1:	Weather Station Ranges and Tolerances	2
Table 2.1:	Comparison Between Bluetooth Low Energy and Bluetooth Classic	8
Table 2.2:	Specifications of The JDY-08 BLE Module	9
Table 5.1:	Wind Speed Measurements Produced by the Wind Speed Sensor and the Wind Speed Meter	49
Table 5.2:	Temperature Measurements Produced by the BME280 Temperature Sensor and the Fluke 52 Thermometer.....	54
Table 5.3:	Humidity Measurements Produced by the BME280 Sensor and the Weather Station Standard	57
Table 5.4:	Pressure Measurements Produced by the BME280 Sensor and the Weather Station Standard	59
Table 5.5:	Rain Measurements Produced by the Rain Sensor and the Calculated Standard	62
Table 5.6:	Photon's RTC Time Vs Blynk's RTC Time.....	74

LIST OF CONSTANTS AND ABBREVIATIONS

Ah	Ampere hours	
ADC	Analog to Digital Converter	
App	Application	
BLE	Bluetooth Low Energy	
°C	Degree Celsius	
E	East	
g	earth-surface gravitational acceleration	9.80665 m/s ²
GPIOs	general-purpose input/output	
GB	Gigabyte	
hPa	hectopascal	
h	hours	
I2C	Inter-integrated Circuit	
kb	kilobyte	
km/hR	kilometres per hour	
kΩ	kiloohm	
LED	Light Emitting Diode	
L	temperature lapse rate	0.0065 K/m
Li-ion	Lithium Ion	
LDO	Low dropout	
m	Metres	
μA	Microamp	
μC	Microcontroller	
M	molar mass of dry air	0.0289644 kg/mol
mA	Milliamp	
mAh	Milliamp Hour	
ml	Millilitre	
mm	Millimetres	
mm/day	Millimetres per day	
mm/hR	Millimetres per hour	
mV	millivolts	
N	North	
NE	North East	

NW	North West	
Ω	Ohm	
Pa	Pascal	
ABS-PC	POLYCARBONATE/ABS ALLOY (PC/ABS)	
PSU	Power Supply Unit	
PCB	Printed Circuit Board	
R	ideal (universal) gas constant	8.31447 J/(mol·K)
SPI	Serial Peripheral Interface	
SE	South East	
S	South	
SW	South West	
UART	Universal Asynchronous Receiver/Transmitter	
UV	Ultra Violet	
V	Volt(s)	
W	West	

CHAPTER 1: INTRODUCTION

Small scale weather stations are useful in applications across various industries that require localised real time weather condition monitoring: agricultural applications where monitoring of temperature aids in irrigation control or in construction where wind speed must be considered for employee safety when working at heights or in aviation which requires the monitoring of various weather variables to ensure safe airplane landings and take-offs. This report describes the methods and processes that were involved in engineering a low power portable weather station which, unlike most other weather stations that use Wi-Fi or GSM communication medium, communicates to a mobile device over Bluetooth. A portable weather station removes the need for a continuous connection to a power outlet and allows the station to be temporarily setup in areas where a local supply is not available. Furthermore, since Bluetooth is used instead of Wi-Fi or GSM, the station's power requirements are comparatively reduced which enables the station to achieve a longer period of operation [1]. Using a mobile device for the interface eliminates the need for a separate specialised display which reduces cost and is a convenient method of interaction with the station.

1.1 Design Specifications

The proposed weather station is required to measure temperature, pressure, relative humidity (humidity), wind speed, wind direction and rainfall. Since this weather station is required to be operated in South Africa's climate, it was logical to determine each weather variable's range by considering the extreme weather conditions experienced in South Africa [2][3]. The extreme temperature and wind speed conditions, -20.1 to 50 °C and 0 to 186 km/hr respectively, were readily available but the remaining variables had to be determined as follows:

- The highest daily rainfall rate experienced in South Africa, 597 mm/day, was converted to a maximum hourly rainfall rate of 24.88 mm/hr which was used to determine the upper limit of the rainfall range;
- The relationship between the altitude above sea level and pressure was exploited to determine the upper and lower limit of the pressure range. The highest and lowest altitudes in South Africa were applied to the equation 1-1:

$$P = P_0 \times (1 - 2.25577 \times 10^{-5} \times h)^{5.25588} \quad [1-1]$$

where P is the pressure in Pa, P_0 is the pressure at sea level in Pa and h is the altitude in metres above sea level [4]. Since the highest point in South Africa is 3450 m above sea level [5] and the lowest point is sea level (0 m), the resulting pressure at each point was calculated to be 661.88 hPa and 1013.25 hPa respectively. To account for pressure variations that occur at these points, mostly due to seasonal changes in temperature and moisture conditions [6], a pressure range below and above these thresholds, 600 hPa and 1050 hPa, was chosen;

- Relative humidity is measured as a percentage has a range of 0-100%;
- 8 standard cardinal directions: North, North East, East, South East, South, South West, West and North West were chosen.

The ranges that were selected, as well as the extreme weather conditions that informed them, are summarised in the table shown in **Table 1.1**.

Weather variable	Extreme weather range required	Proposed weather station range	Tolerance
Temperature	-20.1 to 50 °C	-22 to 55 °C	± 2 °C
Pressure	661.88 to 1013.25 hPa	600 to 1050 hPa	± 10 hPa
Relative humidity	0 to 100%	0 to 100%	$\pm 5\%$
Wind speed	0-186 km/hr	0 to 190 km/hr	$\pm 10\%$
Wind direction	N, NE, E, SE, S, SW, W, NW	N, NE, E, SE, S, SW, W, NW	$\pm 5^\circ$
Rainfall	0-24.88mm/hr	0 to 28 mm/hr	± 2 mm/hr

Table 1.1: Weather Station Ranges and Tolerances

The tolerances specified for each variable in **Table 1.1** were determined by the consideration of the tolerances specified by other standard weather stations [7] and the electronic sensors available in the market.

Since the weather station relies on Bluetooth communication, a user is required to remain in the vicinity of the weather station in order to observe the readings on the mobile device. Hence, it was deemed reasonable for a portable weather station of this nature, which is meant to be temporarily setup at required locations and to be run for short periods of time, to achieve an operating period of at least 12 hours.

1.2 Project Process

The process described below was implemented:

- A time management schedule (shown in Annexure A) was developed which describes the breakdown and time allocation of each task;
- The weather variables and their required ranges as well as the operating period of the proposed weather station were determined;
- Components that were required to satisfy the design requirements were selected and procured;
- Circuits were constructed and the microcontroller (Photon) was programmed;
- Sensors and components were tested against standards and calibrated accordingly;
- The PCBs were designed and sent for manufacturing;
- The mobile application platform was selected and the weather station interface was designed and configured;
- The weather station's functionality was verified;
- A suitable enclosure was selected and modifications were made to accommodate the weather station's circuitry;
- The circuitry was installed into the enclosure and final functional tests were completed.

1.3 Project Limitations

The following design limitations are applicable:

- Although the weather station hardware communicates via Bluetooth, the Blynk app requires an internet connection to access Blynk's servers for login purposes;
- Since Bluetooth was chosen as the communication medium, weather data can only be viewed on the app when it is located within the Bluetooth transmission distance of the weather station;
- The weather station was designed to measure the weather conditions experienced in South Africa only. It will not be able to be used in other countries if the extreme weather conditions of those countries lie outside of the weather station's measuring range;
- Since the weather station is powered by a battery, its period of operation is limited by the battery's capacity;

- The rain sensor will not be able to detect rainfall that lasts for a duration that is less than the evaluation period (15 minutes) and that has a rate less than 0.8 mm/hr;
- Although the weather station was designed to measure temperatures up to 55 °C, the BGT WS-601ABS2 rain sensor will not be able to produce reliable rainfall measurements in environments up to this temperature because it has a maximum operating temperature of 50 °C;
- Although the weather station was designed to operate under temperature conditions that range from -22 °C to 55 °C, the Li-ion battery has an operating temperature range of -20 °C to 55 °C. Since this battery is housed separately from the temperature sensor, care must be taken not to expose it to the full operating temperature range of the weather station;
- Wind speeds that are less than 0.905 km/hr will not be detected by the Davis wind speed sensor since that is the minimum speed measurable by the sensor;
- The wind direction sensor needs to be aligned to face North when it is setup at a new location. Therefore, the North direction needs to be known to allow the sensor to produce correct measurements;
- The micro SD card module is only compatible with micro SD cards up to 32 GB in size which limits the number of data entries that can be logged onto the SD card. Since each log file is 0.95 MB, the maximum number of files that can be logged to a 32 GB SD card is 33 684;
- Any future hardware additions to the design may not drive the current consumption of the weather station beyond the 200 mA maximum that is deliverable by the charge controller.

CHAPTER 2: DESIGN

This chapter attempts to outline the elements that are needed and the components that were selected to fulfil the design requirements of the portable weather station project.

2.1 Block Diagram

The block diagram, shown in **Figure 2.1**, outlines the major components that are required for the design of the proposed weather monitoring system.

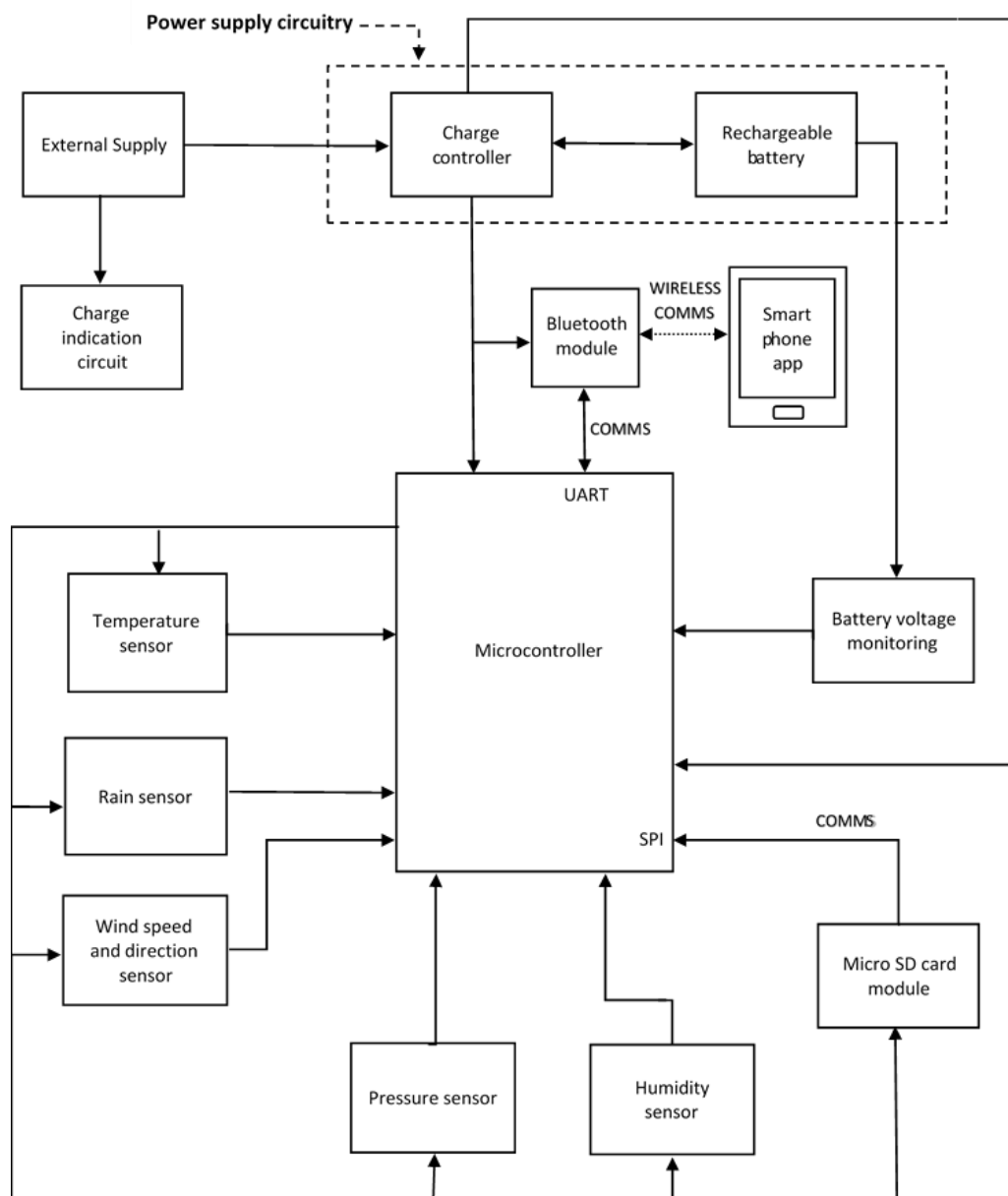


Figure 2.1: Block Diagram of the System

2.1.1. The Photon

The STM32 ARM Cortex M3, contained in the Particle Photon development board (Photon), was selected as the μC of choice due the fact that it satisfied and exceeded the design requirements as follows [8]:

- It has a wide selection of GPIOs which include analogue, digital, I2C, SPI and UART pins for Bluetooth communication (see **Figure 2.2**);
- The Photon has a low current consumption of 80 mA which can be further reduced by cycling through sleep modes;
- The Photon has a small physical profile with dimensions of 36.58 mm by 20.32 mm;
- The Photon's inbuilt ADC has a 12-bit resolution which allows it to measure voltages in increments of 0.8 mV;
- It has adequate online support and documentation.

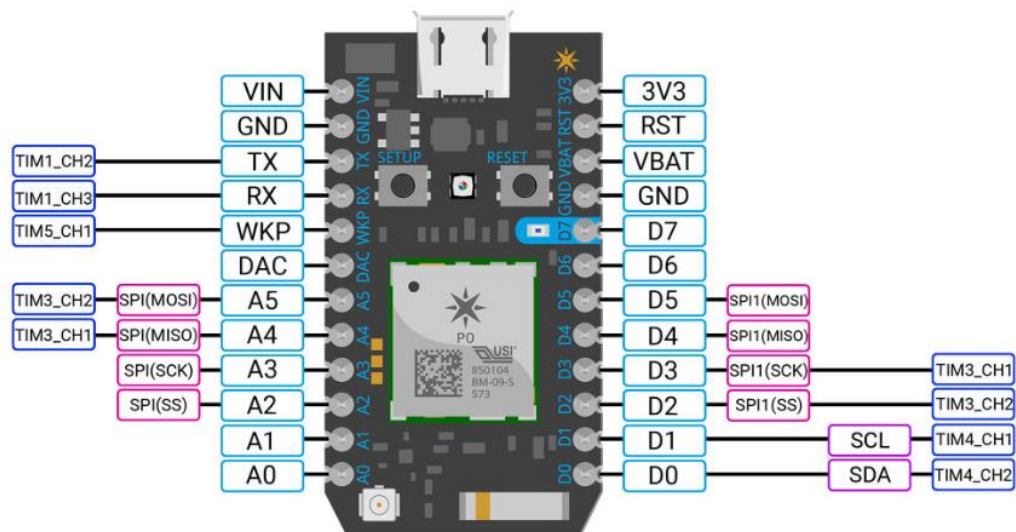


Figure 2.2: Pinout of the Particle Photon [9]

2.1.2. Temperature, Pressure and Humidity Sensor

The Bosch BME280 was selected because it is a 3-in-1 temperature, pressure and humidity sensor that [10]:

- Is maintenance free;
- Is compatible with the Photon's 3.3 V supply.
- Can operate throughout the measuring ranges, and within the tolerances, required by the design specification:
 - Temperature: $-40\text{ }^{\circ}\text{C}$ to $+85\text{ }^{\circ}\text{C} \pm 1.25\text{ }^{\circ}\text{C}$;

- Pressure: 300 hPa to 1100 hPa ± 1.7 hPa;
- Humidity: 0 to 100% $\pm 3\%$.
- Has a low 3.6 μ A current consumption;
- Is combined into a small physical profile with dimensions 21 x 13 mm which reduces the amount of space required;
- Reduces the number of connections and GPIOs needed from the Photon (2 connections for power and 2 input pins) compared to the number of wires and input pins required by 3 separate sensors;
- Operates over I2C which affords it the following advantages over analogue sensors:
 - It is addressable and therefore its connection to the weather station can be detected without the need of additional circuitry;
 - More than one I2C sensor can be connected to the Photon, simultaneously, using the same set of input pins; this reduces the number of GPIOs required from the Photon.

2.1.3. *Wind Speed and Direction Sensor*

A second-hand Davis 6410 electromechanical wind speed and direction sensor, that was available from the Durban University of Technology, was found to be appropriate for this design because it [11]:

- Has a 0-322 km/hr and 0°-360° wind speed and direction range;
- Has $\pm 5\%$ and $\pm 3^\circ$ wind speed and direction tolerances;
- Is compatible with the Photon's 3.3 V supply;
- Has an operating temperature range of -40 °C to 80 °C, making it suitable to be operated under the extreme temperature conditions experienced in South Africa;
- Was built with durable and treated material to make it resistant to environmental conditions, such as UV radiation and rain;
- Has a low 165 μ A current consumption (according to Ohm's Law, since the sensor consists of a 20 k Ω potentiometer with a 3.3 V potential difference across it [12] [13]);
- Includes a 12 m, insulated and water proof, cable;
- Requires minimal maintenance: the wind sensor's pointer arm may drift from its set orientation, over time, and is required to be checked annually [14].

2.1.4. *Rain Sensor*

A BGT WS-601ABS2, electromechanical tipping bucket type, rain sensor was selected to measure rainfall. This was selected because it [15]:

- Has a measuring range of 0-240 mm/hr which is well above the 0-28 mm/hr required by this design;
- Has an adequate $\pm 4\%$ measurement tolerance which is lower than the $\pm 6\%$ tolerance required by the design specification;
- Is compatible with the Photon's 3.3 V supply and, due to the reed switch mechanism of operation, has a low μA -level current consumption;
- Is made with strong ABS-PC enabling it to withstand environmental conditions;
- Has a resolution of 6.27 ml, which corresponds to 0.2 mm of rainfall;
- Was 30% cheaper than the nearest competitor;
- Has low maintenance: this sensor is susceptible to becoming clogged by falling debris and it is recommended that the sensor be cleaned bi-annually.

2.1.5. *Bluetooth Module*

Bluetooth was the medium of communication specified, in the design, to interface the Photon with the mobile app. It was chosen due to its low power requirements and low cost factor (low price modules and no data transmission costs) when compared to other wireless communication methods such as Wi-Fi or GSM.

Bluetooth technology can be broken down into two broad categories: Bluetooth Low Energy (BLE) and Bluetooth Classic. These were investigated to determine which category best suited the design requirements and the comparison is shown in **Table 2.1** [16].

	Bluetooth Low Energy (BLE)	Bluetooth Classic
Physical profile	Small	Small
Power requirement	In the micro-amps range	In the milliamps range
Transmission range	Medium to short distance	Short distance
Initial Cost	Low	Medium
Data throughput	Low	Medium

Availability	Locally available	Locally available
---------------------	-------------------	-------------------

Table 2.1: Comparison Between Bluetooth Low Energy and Bluetooth Classic

BLE has a greater transmission distance, significantly lower power consumption and its data throughput rate, although smaller than Bluetooth Classic, is adequate for the communication of sensor data. Therefore, it was chosen for this project.

Table 2.2 shows the key specifications of the BLE module that was chosen [17].

Physical profile	44 mm x 19 mm
Power requirement	8.5 mA
Transmission Range	0-30 m
Price	R94.95
Availability	Locally available
Operating voltage	3.3 V
Operating temperature range	-40 °C to +80 °C

Table 2.2: Specifications of the JDY-08 BLE Module

2.1.6. Data Logging using a Micro SD Card Module

Since the weather station's data communication was localised, data logging was achieved using a standard 4 GB micro SD card and a micro SD card reader. The Robotdyn SD card reader, which is based on the module designed by Adafruit, was selected because it [18]:

- Operates on the 3.3 V supplied by the Photon;
- Has a small physical profile (36 mm x 25 mm);
- Has a low current consumption (5 mA);
- Communicates over SPI;
- Has a dedicated card detector pin which simplified the detection of the SD card.

2.1.7. Power Supply Circuitry: Rechargeable Battery

After investigating the different types of rechargeable batteries that were available, it was determined that a 3.7 V rechargeable Li-ion battery was the most suitable option for this design project:

- All the components that constitute the weather station (i.e. the Photon, Bluetooth module, the SD card module and the sensors) can be powered from a single Li-ion cell with this voltage rating;
- Li-ion batteries have [19]:
 - Many charge/discharge cycles;
 - High energy efficiency;
 - High energy-to-weight ratios i.e. they are lightweight when compared to other battery types with similar electrical ratings;
 - Large capacities that fit into small profiles which is ideal for portable applications;
 - A fast charging ability;
 - Low self-discharge rates;
 - High availability due to their popularity.

The battery capacity that was needed was determined by considering the:

- Weather station's required hours of operation;
- Total current consumption of the weather station.

The total current consumption was only able to be established at a later stage, when all the electronic hardware of the weather station was connected. This was measured to be 69 mA at the time when the battery capacity was being determined. Once both the current consumption and period of operation was known, equation 2-1 was applied [20]:

$$\text{Period of Operation (h)} = \frac{\text{Battery Capacity (Ah)}}{\text{Current Consumption(A)}} \quad [2-1]$$

Therefore, the required battery capacity was calculated to be 828 mAh. A rechargeable 1050 mAh Li-ion battery was the next nearest rating that satisfied this requirement. The 1050 mAh 3.7 V 063450AR Li-ion battery was selected because it [21]:

- Has over voltage and under voltage protection, preventing the battery from being operated under damaging and dangerous conditions;
- Was available at a local store.

2.1.8. Power Supply Circuitry: Charge Controller

Li-ion batteries have a low tolerance to overcharging. Overcharging diminishes their capacity and can cause them to combust. Therefore, a reliable charge controller was needed to ensure safe charging of the 3.7 V 1050 mAh Li-ion cell. The Adafruit USB Lilon/LiPoly charger - v1.2, which is based on the MCP73833 linear lithium ion/lithium polymer charge management controller, was selected because it [22]:

- Was designed to safely charge a single 3.7 V lithium ion cell with a regulated voltage output of 4.2 V ($\pm 0.75\%$) and when the charge current diminishes to 5% of the regulated charge current (I_{REG}), the charge cycle is automatically terminated to prevent over charging of the cell;
- Has an input voltage rating of 5 V which is convenient because it allows the battery to be charged by cell phone and tablet chargers or power banks;
- Has the ability to simultaneously charge the Li-ion cell and deliver power to the load;
- Has two charge indicating STAT pins that can be interfaced with a Photon for charge monitoring purposes;
- Is compact in size (33 mm x 35 mm x 7 mm) and light weight (5.7 g) which makes it convenient for portable applications.

2.1.9. Battery Status Monitoring Feature

This feature was achieved by using two separate circuits: a voltage divider circuit was used to monitor the battery voltage and the charge controller's STAT1 and STAT2 pins were used to monitor the charge status of the battery. Since the charge controller pins were readily available no additional components were required for charge status monitoring. However, standard 68 k Ω and 33 k Ω resistors were selected to form the voltage divider circuit which scales down the battery voltage applied to the Photon's ADC. These components are used to achieve the battery voltage monitoring feature because:

- They form a simple and effective solution;
- The combined 101 k Ω resistance was the resistance suggested by the developers of the Particle Photon [23];
- When the battery's 4.2 V maximum is applied to the divider circuit, the highest voltage across the 68 k Ω resistor would be limited to 2.828 V, which is within the ADC's 3.3 V limit;

- The maximum current consumed by the 101 k Ω circuit, which would occur when the battery voltage is at 4.2 V, would only be 41.6 μ A. Such a current draw would not significantly impact the power consumption of the project.

2.1.10. Charge Indication LED

The purpose of this circuit is to provide an indication when an energised charger is connected to the weather station, without needing to consult the app. The circuit consists of a standard 5 mm yellow light emitting diode (LED) and 220 Ω current limiting resistor.

2.1.11. Connector and Cable Selection

The use of connectors affords this weather station the flexibility of a modular design ie. sensors can be connected and disconnected from the weather station as required. The RJ45 standard was used for all the weather station's sensors [24].

USB-C was chosen as the power connector since it is fast becoming the standard for both power and data transmission applications [25]. It will be used to interface any standard 5 V USB-C charger to the charge controller. The added benefit of choosing a different connector for power, compared to the one chosen for the sensors, is that it prevents the power cable from accidentally being applied to the Photon's GPIO pins. The SparkFun USB-C breakout board was chosen because:

- It removed the difficulty of soldering the small and thinly spaced pins of the USB-C connector onto a PCB;
- It was locally available.

Standard Cat-5e cable was selected as the medium to link the BME280 and rain sensors to the weather station due to its flexibility as well as resistance to rain and UV radiation. The wind sensor already included its own UV and rain proof 4 core cable. All the cables were terminated using the T568A wiring standard [26].

2.1.12. Mobile App

The weather station app was created using the Blynk platform because it [27]:

- Easily interfaces with the Particle Photon;

- Simplifies the communication with the Photon using customizable widgets;
- Allows for the quick and easy design of apps due to its drag-and-drop building style;
- Has support for both iOS and Android mobile platforms;
- Is free to download and credits, used to buy widgets, can be purchased at cheap rates as required;
- Allows the app designer to publish the app to the app stores as a secure standalone app that is customized with the designer's information. Although this is not a requirement for this design, the option is available should it become necessary in the future.

CHAPTER 3: BUILD

The circuit configuration and wiring of each hardware element of this design project as well as the build of the mobile application is discussed in this chapter.

3.1 The Spare I2C Port

The purpose of this port is to future proof the weather station. An RJ45 connector was used to form a high power I2C port. The port connections are in a logical sequence: the charge controller's +V and GND pins are connected to pin 1 and pin 2 of the RJ45 port while the Photon's D1 and D0 pins connect to pin 3 and 4 of the RJ45 port respectively. The circuit diagram is shown in **Figure 3.1**.

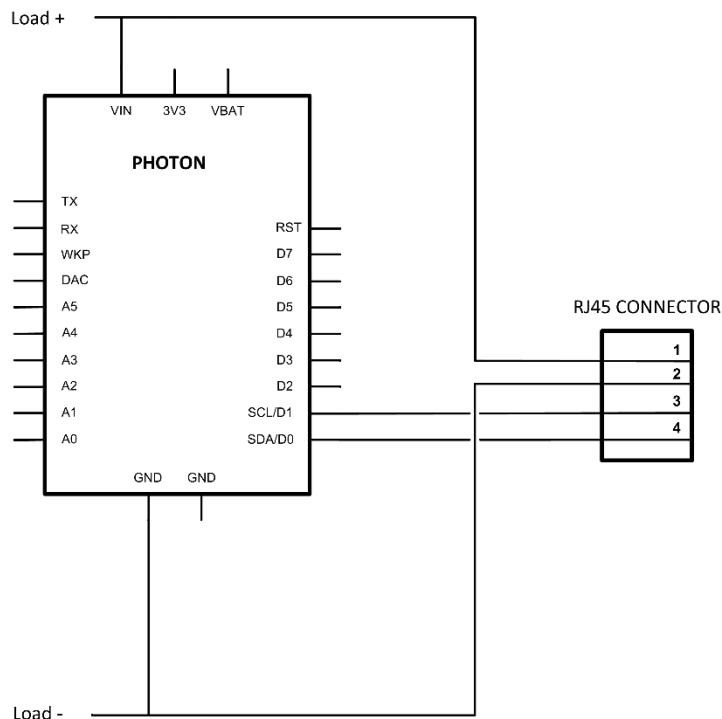


Figure 3.1: Connection of the Spare I2C Port

Since this I2C port is powered directly by the charge controller, it can provide a higher current output (200 mA total) than the Photon's 3V3 supply (100 mA total). This allows the weather station to accommodate:

- Higher power I2C devices;
- More I2C devices, with the use of a suitable adapter.

3.2 The BME280 Temperature, Humidity and Pressure Sensor

The BME280 sensor is powered directly by the Photon's 3.3 V supply pin because its input supply tolerance is only 3.3V to 3.6 V. However, since the BME280 is an I2C sensor, its SDA and SCL pins are connected to the Photon's D1 and D0 pins via pins 3 and 4 of its RJ45 port. The circuit diagram is shown in **Figure 3.2**.

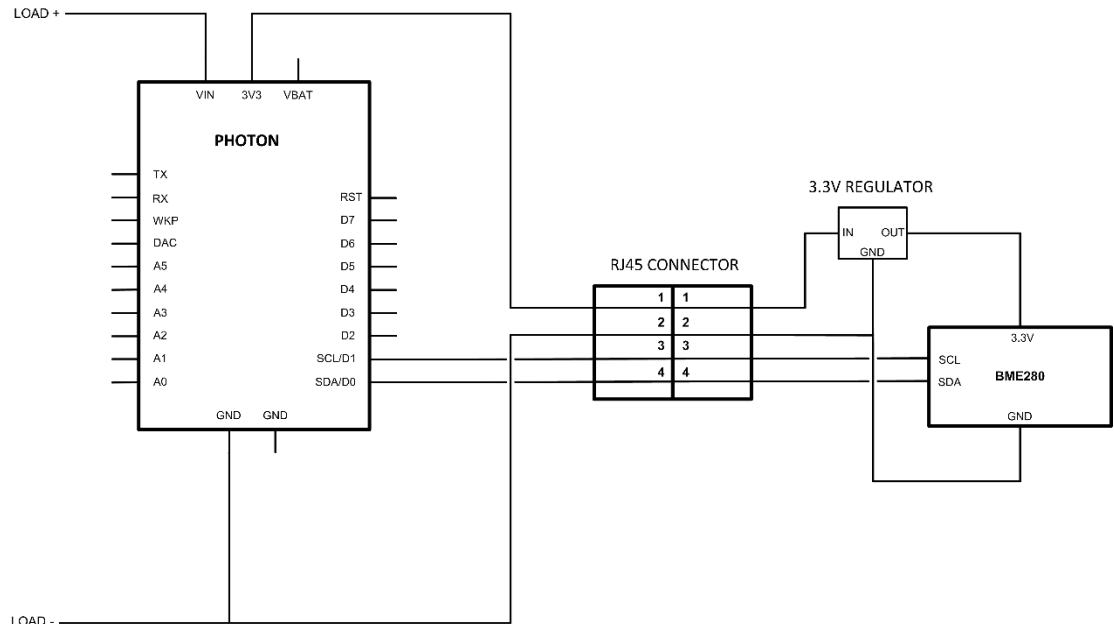


Figure 3.2: Connection of the BME280 to the Photon

A LDO 3.3 V regulator was coupled to the sensor to protect it when it is connected to the higher voltage I2C port [28]. Since an LDO regulator was used, the operation of the sensor is unaffected when connected to the 3V3 supply. The SparkFun BME280 breakout board has on board pull-up resistors, so no external pull-up resistors were required.

3.3 Wind Sensor

3.3.1 Repairs

The second-hand Davis 6410 wind sensor needed to be repaired before it could be used in this design project. The anemometer's mechanism was slightly rusted and contained debris which prevented it from rotating smoothly. The head of the sensor was carefully cut in half,

the debris was removed, the moving parts were lubricated and the reed switch was also replaced [29]. Before resealing the head with a strong adhesive and marine silicone (to maintain water proofing), the sensor was connected to the Photon and tested to ensure it was functional.

3.3.2 Wiring and Circuit Diagram

This sensor's wind speed and direction circuit as well as its detection circuit is powered directly from the Photon. The 3.3 V supply is required to power this sensor because the output voltage from the wind direction circuit is measured by an ADC pin that has a maximum input voltage of 3.3 V. The outputs from the wind speed and direction sensors are connected to pins D3 and A1, respectively. The input and output terminals of the detection circuit are connected to the 3V3 and WKP (which is configured as digital input) pins, respectively. No external pull-up resistors were implemented because the Photon's internal resistors were used to prevent pins D3 and A4 from "floating". All 6 wires of the wind sensor were connected to the Photon via the RJ45 connector as shown in **Figure 3.3**.

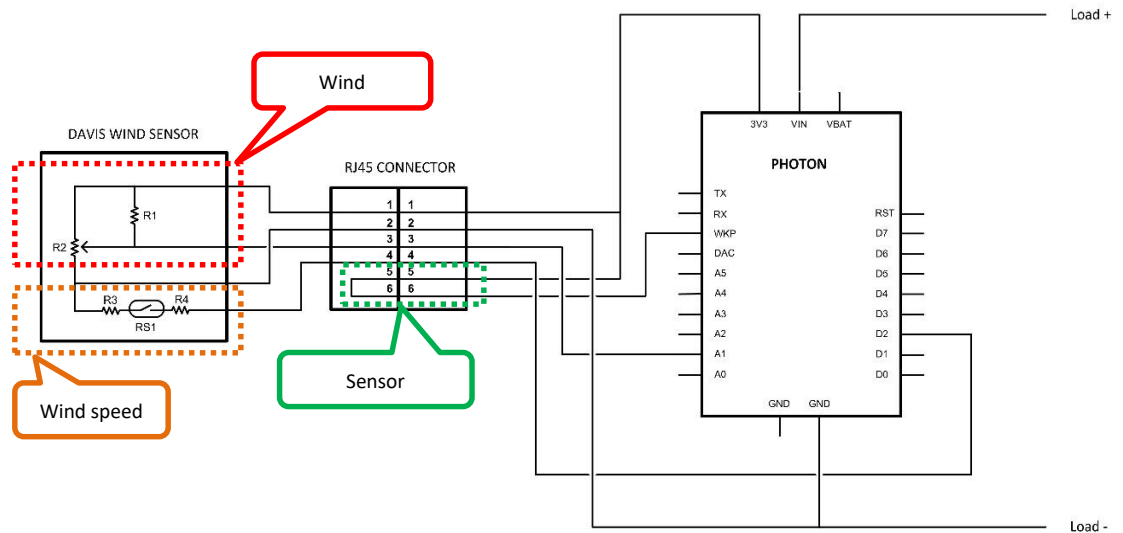


Figure 3.3: Connection of the Davis 6410 Wind Sensor to the Photon

The wind speed sensor mechanism consists of 3 cups attached to a base with a magnet and a reed switch (RS1). Resistors R3 and R4 protect the reed switch in the event that power is applied to the wrong terminals [30]. When the wind blows, the cups rotate and the magnet closes the reed switch once per rotation. Each time the switch closes, the voltage applied to D2 pulses low. This change in state is used to measure the wind speed.

The wind direction sensor consists of a continuous rotation potentiometer (R2) attached to a wind vane that points in the direction of the wind. The resistance of R2 changes according to the position of the wind vane which in turn changes R2's output voltage. This voltage is applied to pin A1 and is used to determine the wind direction. Resistor R1 prevents R2's output from floating when its wiper is crossing the open end points of its resistance.

3.4 Rain Sensor

The Photon's 3.3 V supply forms the input for the rain sensor's measuring and detection circuits. The detection and rain sensor outputs are connected to the Photon's D5 and D6 pins, respectively, via an RJ45 connector. The circuit diagram is shown in **Figure 3.4**.

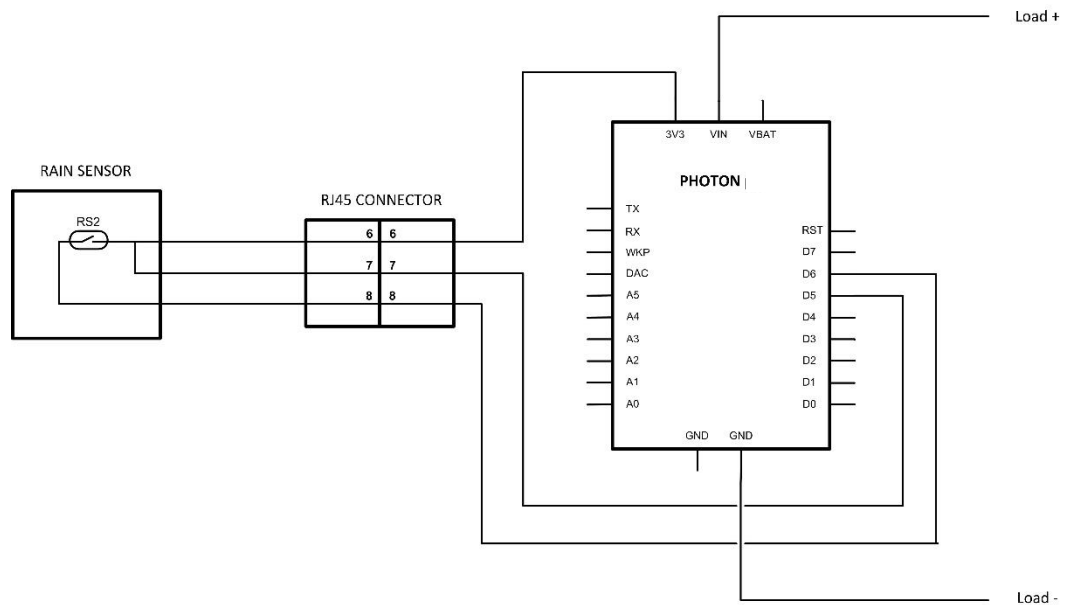


Figure 3.4: Connection of the BGT WS-601ABS2 Rain Sensor to the Photon

RJ45 pins 6-8 were used in order to avoid a short circuit condition from occurring if this sensor is mistakenly connected to any of the other weather station ports: the positive 3V3 supply will be connected to the ground connection on pin 2 of the other ports if the RJ45 pins 1-3 were used for this sensor.

This sensor consists of 2 buckets attached to a see-saw pivot mechanism, a magnet and an enclosed reed switch (RS2). Refer to **Figure 3.5**.

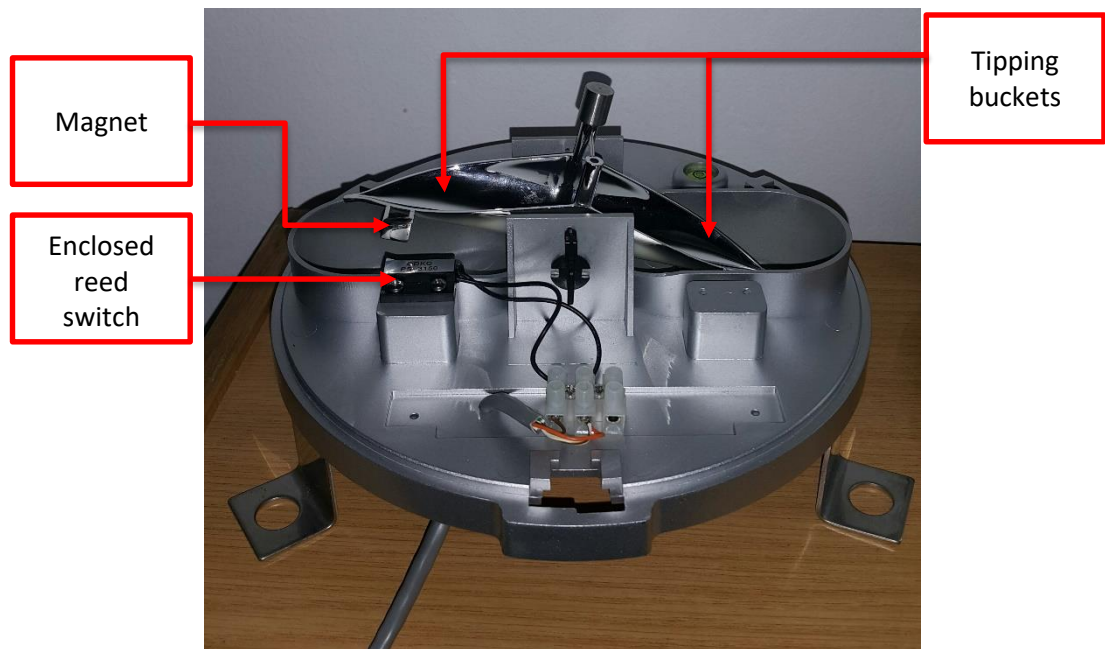


Figure 3.5: Tipping Bucket Mechanism

Each time one of the sensor's two buckets fills up with sufficient water (6.27 ml), that bucket tips over to empty out its contents and simultaneously positions the other bucket to fill with water. With each tipping motion the magnet attached to one of the buckets passes over the enclosed reed switch, causing it to close momentarily. The closing of the reed switch causes a pulse that is detected by the Photon.

3.5 Bluetooth

Since the Bluetooth module is 5 V tolerant, it is energised directly by the load outputs of the charge controller in order to reduce the power drawn from the Photon. The TX and RX pins of the module are connected to the Photon's RX and TX UART serial communication pins. The circuit diagram is shown in **Figure 3.6**.

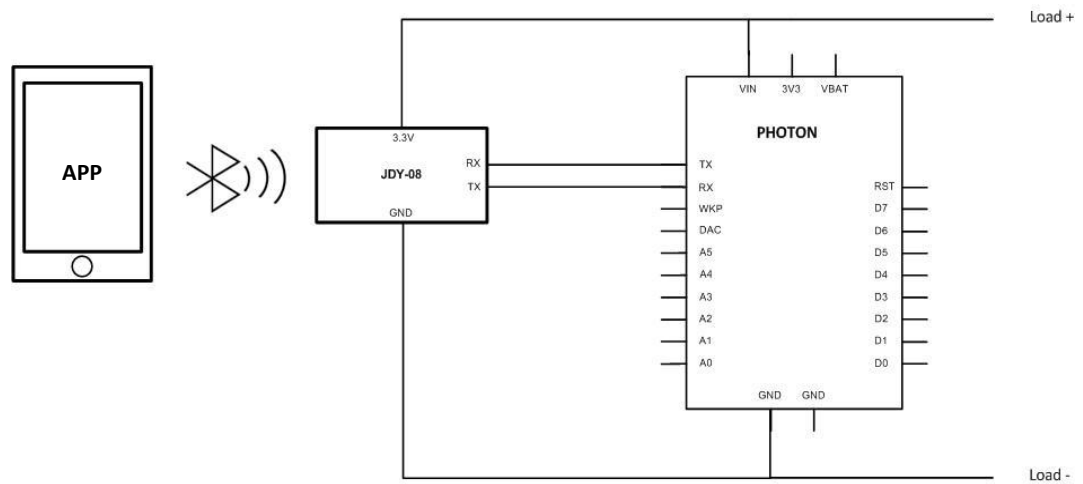


Figure 3.6: Connection of the Bluetooth Module to the Photon

3.6 Micro SD Card Module

The micro SD card module communicates with Photon over SPI and is powered directly from the charge controller's load outputs (**Figure 3.7**), because SD card readers are prone to current spikes that would exceed the 100 mA limit of the Photon's 3.3 V supply [31].

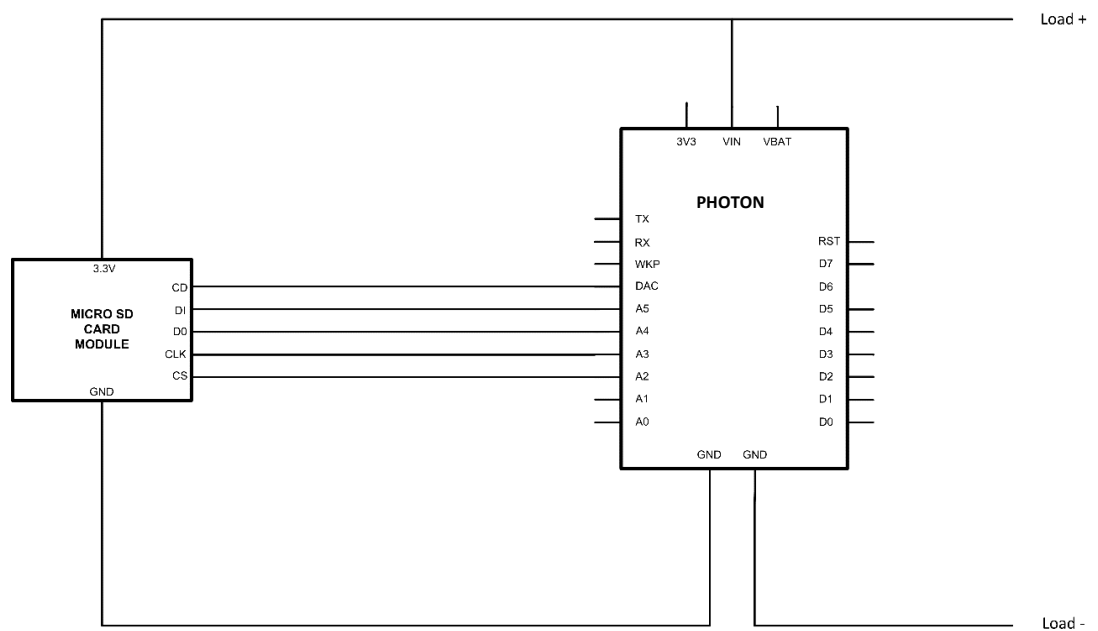


Figure 3.7: Connection of the Micro SD Card Module to the Photon

3.7 Power Supply Circuitry

The Li-ion battery is connected directly to the charge controller while the load is connected via a switch. This allows the battery to be charged, regardless of the switch's position. The circuit diagram is shown in **Figure 3.8**.

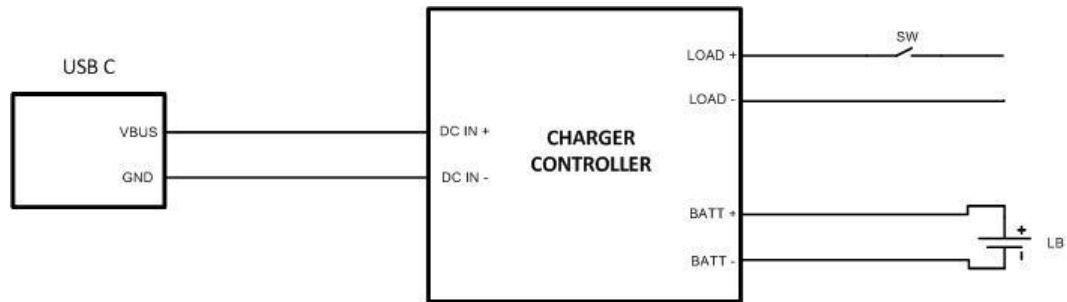


Figure 3.8: Charging Circuit

3.8 Battery Status Monitoring

The STAT1 and STAT2 pins of the charge controller are connected directly to the Photon's D3 and D4 pins, respectively. The voltage across the 68 kΩ resistor is measured by the Photon's ADC on pin A0. The circuit diagram is shown in **Figure 3.9**.

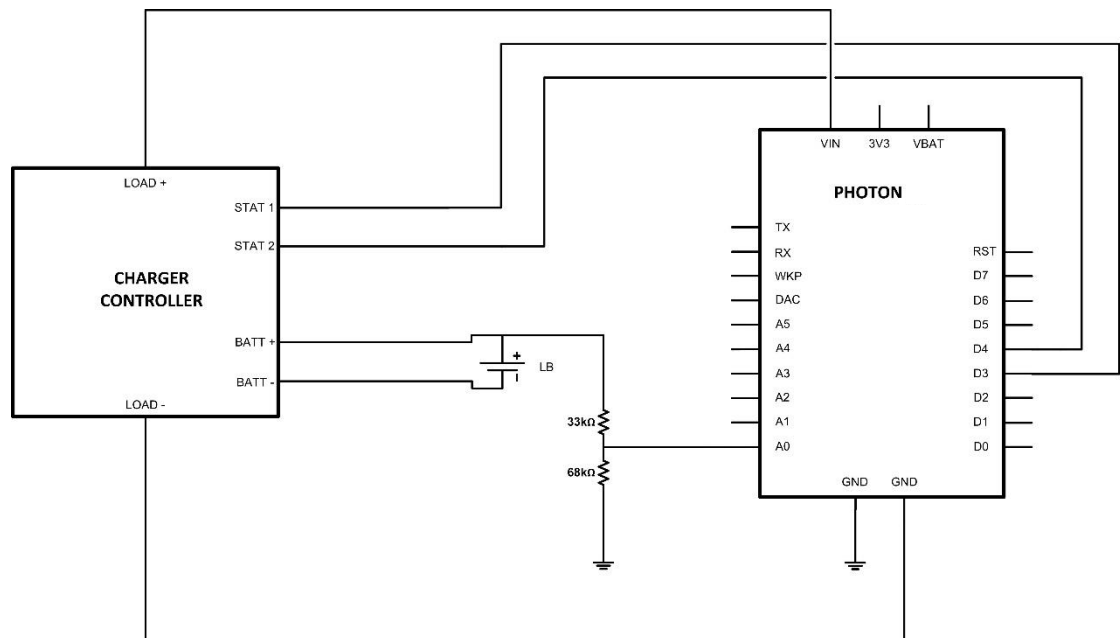


Figure 3.9: Circuits used to Achieve the Battery Status Indication Feature

3.9 Charge Indication Circuit

The LED and 220 Ω current limiting resistor combination, which was measured to limit the current to 11.04 mA, is connected in parallel to the charge controller's inputs as shown in **Figure 3.10**.

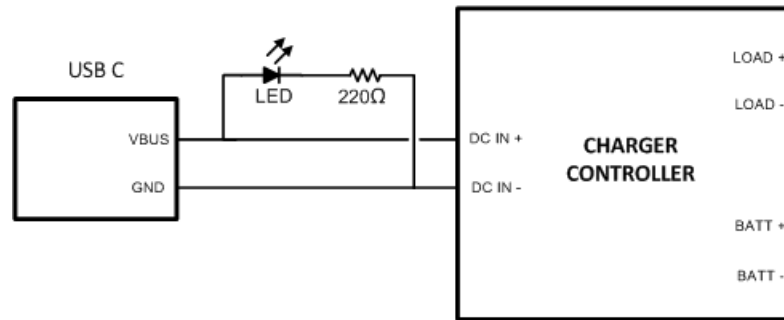


Figure 3.10: Charge Indication Circuit

3.10 Design of the PCB

The PCBs were designed according to the completed weather station circuit diagram shown in Annexure B. This project required 2 separate PCBs:

- The main PCB (dimensions 132 mm x 95 mm) which consists of the Photon, charge controller, indication circuitry, Bluetooth module, micro SD card module, USB-C port, switch and RJ45 ports;
- The BME280's PCB (dimensions 46 mm x 38 mm) which consist of a 3.3 V regulator, the BME280 module and an RJ45 port. The BME280 required its own PCB so that it could be housed separately from the other components of the weather station or else the heat generated by the other components would cause false temperature readings.

Both PCBs were designed using the online tool suite EasyEDA which was selected since it:

- Is free and simple to use;
- Provides the option to print the designed PCB for a significantly (80%) lower price than other PCB manufacturers that were approached locally.

The following method was implemented to design the PCBs [32][33][34]:

- Components were set up such that those with the most number of connections, such as the Photon, were centred. The rest of the components were positioned, around them, as close as possible to the terminals that they need to be connected to:
 - The ports and female connectors were found to be the next easiest components to place since the positions they occupy are the most limited;

- Consideration was also given to the orientation of the components as some have parts that need to be accessed easily such as the micro SD card slot of the SD card module.
- The supply rails were routed first, along the periphery of the PCB but within a safe distance from the edge, so that they reach all the components with minimal interference with the components' remaining connections;
- After all connection of the PCB were routed, the design was refined by reducing the:
 - Trace route lengths;
 - Proximity of components;
 - Spacing between trace routes;
 - Dimensions of the PCB.
- The designed PCB was printed on paper first, so that it could be evaluated and verified physically, before it was sent to be manufactured.

The completed PCBs are shown in **Figure 3.11**.

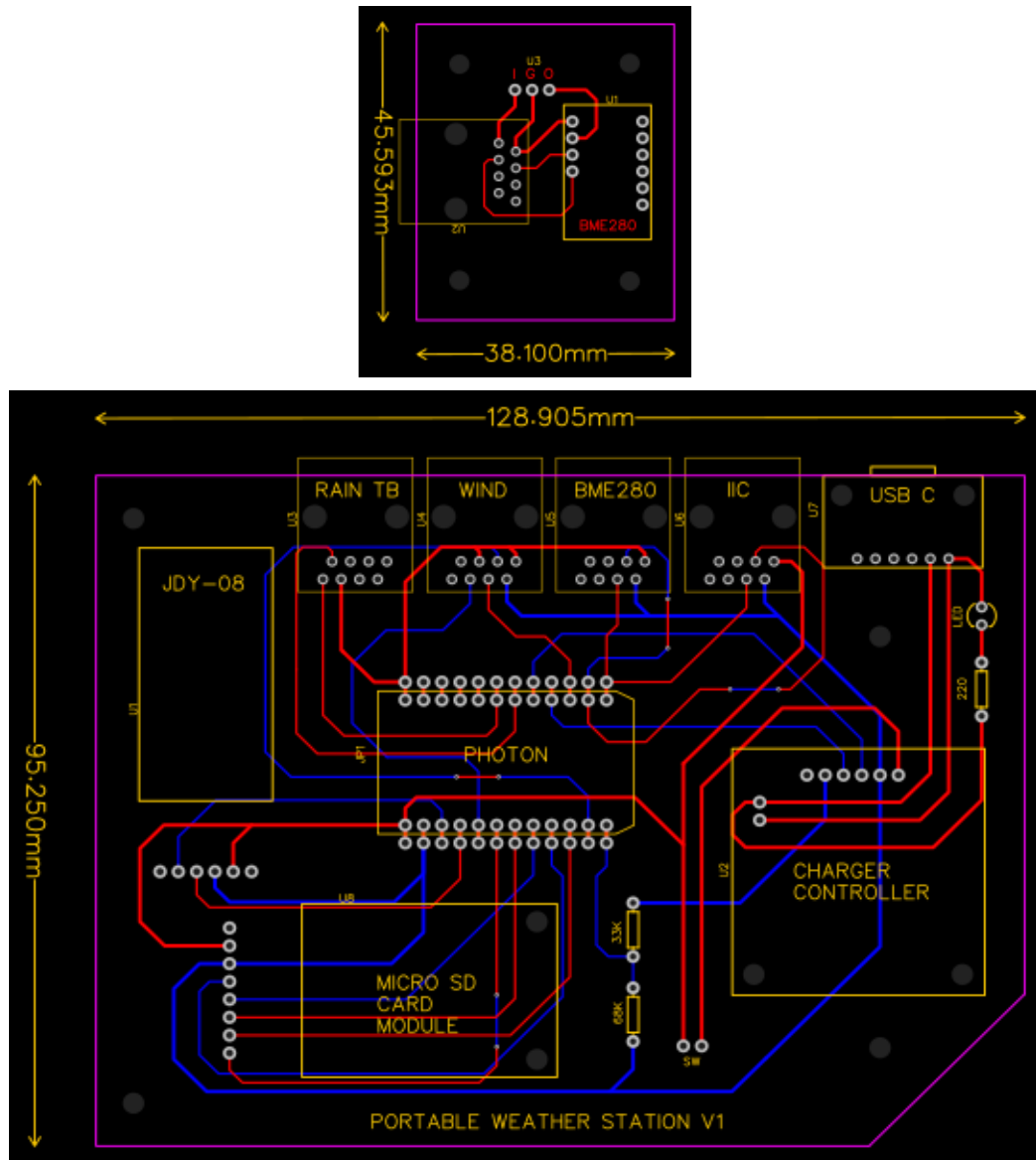


Figure 3.11: Weather Station PCBs

3.11 The Mobile Application

The following were considered when the app interface was designed:

- The font size;
- The colours and type of widgets were selected to make the app aesthetically pleasing;
- Functionality: the widgets were laid out logically so that the interface presents the information in a manner that is easy to understand and is user friendly.

The app interface is separated into 3 categories using the Blynk “Tabs” widget [35]:

1. Live measurements: shows the real time weather measurements on 5 Blynk “Gauge” widgets and 1 “Value Display” widget. The sensor connection statuses are

displayed on 6 “Value Display” widgets and a push button widget is responsible for toggling the data logger;

2. Trends: displays the weather measurement trends over a configurable period on 2 “Superchart” widgets [36];

3. General:

- Indicates the updated battery voltage as well as the battery status conditions;
- The “Bluetooth” widget enables the mobile app to connect to the weather station hardware;
- Consists of 2 non-interactive widgets:
 - a. “Notification” widget which is used to enable push notifications on the mobile device;
 - b. “RTC” widget which allows the weather station hardware access to the Unix time.

An “Image gallery” widget is used to display the name and logo of the weather station. The screenshot of the weather station application interface is shown **Figure 3.12**.

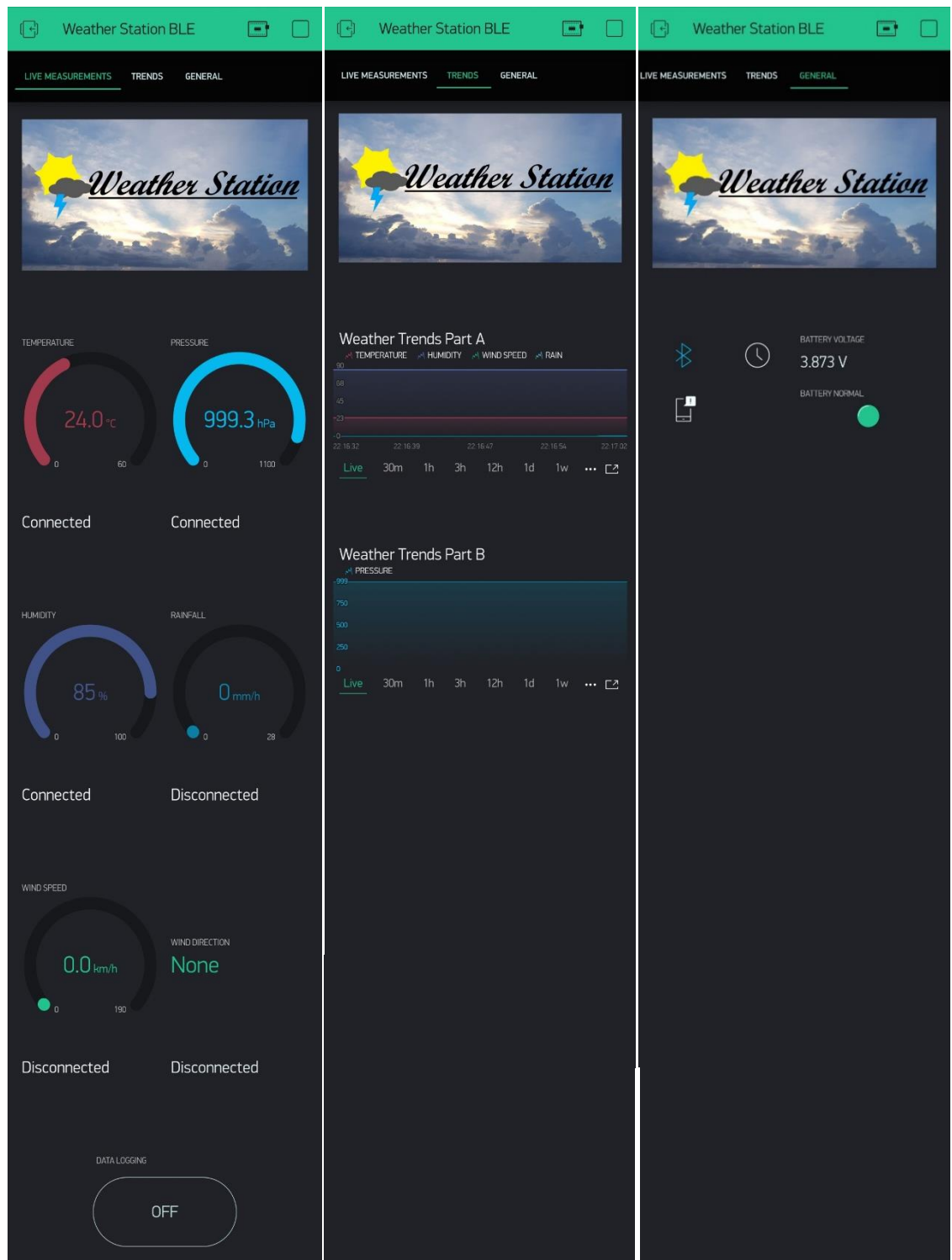


Figure 3.12: Weather Station Application

3.12 Enclosure Selection and Modification

This project focused on the selection and modification of the enclosure intended to house the main PCB. The enclosure with dimensions 174 mm x 107 mm x 29 mm was found to be adequate and was selected by consideration of [37]:

- The dimensions of the main PCB (129 mm x 95 mm) and battery (54 mm x 35 mm) that it is required to house;
- The material makeup of the enclosure:
 - Resistant to deformation;
 - Capable of protecting the circuitry from vibration and impact due to normal use.
- Availability and price.

The enclosure required the following modifications:

- 4 slots were cut out of the left side surface of the enclosure to fit the 4 RJ45 ports and the 1 USB-C port of the PCB;
- An opening was carved out on the right-side surface of the enclosure to accommodate the power switch;
- A 5 mm hole was drilled into the front-left surface of the unit to accommodate the charge indication LED;
- 4 x 3 mm holes were drilled on the bottom surface of the enclosure to accommodate the PCB's mounting screws;
- 3 x 3 mm holes were drilled into the top surface of the enclosure to accommodate 3 strategically placed 12 mm long spacers to secure the Li-ion battery in the enclosure as shown in **Figure 3.13**.

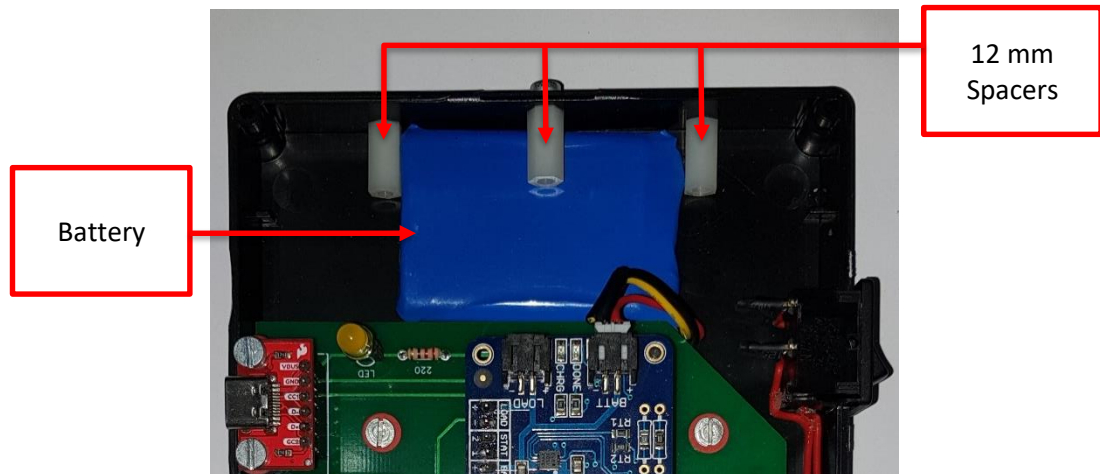


Figure 3.13: Spacers used to Secure Battery in Enclosure

The design of the modified enclosure is shown in Annexure C. The enclosure also includes labelling to indicate the:

- Purpose of each sensor's port which is labelled and outlined in grey except for the high power I2C port which is outlined red to distinguish it from the BME280's I2C port;
- Switch state: "On" and "Off";
- Housing unit name: "Weather Station Central Unit".

CHAPTER 4: FIRMWARE DESIGN

The firmware required to run the weather station was developed in stages. The first stage involved the development of individual firmware programs that enabled the Photon to interact with each major component (the sensors, the app and the SD card module) separately. Once working programs were established, they formed subroutines that constitute the completed weather station firmware that can be found in Annexure D. The purpose of this chapter is to discuss the operation of, and the subroutines responsible for, each element of the firmware design.

4.1. The “void loop()” Subroutine

This subroutine executes repeatedly and consists of only two commands:

- “Blynk.run”, which is necessary for maintaining the connection between the weather station’s hardware and the Blynk app;
- “timer1.run”, which is necessary to maintain the timer.

4.2. The “AllSubroutines()” Subroutine

More lines of code could not be implemented in the “void loop()” subroutine as the chances of encountering Blynk server connection errors increases significantly. Instead, the rest of the code required to operate the weather station was combined into one subroutine, “AllSubroutines”, that is set to execute every 3 seconds by “timer1”, as shown in **Code Listing 4.1**.

```
//Executed, by a timer, every 3 seconds
void AllSubroutines(){

    GetTime(); //Obtain current time and date from Blynk
    Windsensor();
    RainSensor();
    GetBME280SensorData();
    SaveToSDCard(); //Checks if data logging should be enabled or not
    BlynkComms(); //Called last to allow variables to be updated
                  before they are transmitted to the mobile app
    PowerSaveMode(); //Checks if stop mode should be activated or not
    //DisplayOnSerialMonitor(); //For debugging purposes
```

```
}
```

Code Listing 4.1: The “AllSubroutine()” Code Segment

4.3. BME280 Operation and Programming

The “GetBME280SensorData()” subroutine, shown in **Code Listing 4.2**, is responsible for:

- Determining if the sensor is connected to the weather station;
- Obtaining the temperature, pressure and humidity measurements.

When the sensor is not connected to the station, the connection status is updated and the temperature, pressure and humidity measurements are zeroed to prevent false readings.

```
//Temperature, humidity and pressure function
void GetBME280SensorData(){

    //Check if the BME280 sensor is connected to the weather station
    if(!bme.begin(Addr)){    //BME280 sensor is disconnected

        BME280SensorState="Disconnected";
        BME280SensorConnected=0;    //To prevent this 'if' statement from
        Executing continuously
        //Serial.println("\nBME280 sensor not connected");

        //Since sensor is disconnected, zero all measurements to prevent
        false readings
        Temperature = 0;
        Humidity = 0;
        Pressure = 0;

    }
    else if(bme.begin(Addr)){    //BME280 sensor is connected

        //Allow the sensors readings to stabilise before considering the
        measurements
        if(BME280SensorConnected==0){

            BME280SensorState="Initialising...";
            BME280SensorConnected=1    //To prevent this 'if' statement
```



```

        From executing continuously
        //Serial.println("BME280 sensor is initialising...");

    }
    else if(BME280SensorConnected==1){

        BME280SensorState="Connected";
        BME280SensorConnected=2;    //To prevent this 'if' statement
        from executing continuously
        //Serial.println("BME280 sensor is connected and setup is
        complete");

    }
    if(BME280SensorConnected==2){

        delay(30);    //To allow BME280 sensor to initialise
        Pressure = (bme.readPressure()/100);    //Pressure measurements
        in hPa
        Temperature = bme.readTemperature();
        Humidity = bme.readHumidity();

    }

}
}
}

```

Code Listing 4.2: BME280 Sensor Measurements

4.4. Davis Wind Sensor Operation and Programming

The primary “Windsensor()” subroutine, and 4 secondary subroutines, shown in **Code Listing 4.3** are responsible for :

- Initialising and terminating the interrupt responsible for tracking the raw wind speed sensor data;
- Enabling and disabling the timer that is used to calculate wind speed measurements;
- Determining the connection status of the Davis sensor;
- Obtaining the wind speed and direction measurements.

```

//Primary wind sensor subroutine

```

```

void Windsensor(){

    if(digitalRead(DetectPin1) == HIGH){ //Sensor connection detection

        //Setup windsensor
        if (WindsensorConnected == 0){

            WindsensorSetup();
            WindsensorState="Connected";
            //Serial.println("Windsensor is connected and setup is
            complete");

        }
        //If there is no wind speed then the wind direction must be blocked
        if(Speed==0){

            DirectionNumber=0;

        }
        else{

            WindDirection(); //Determine wind direction

        }

    }

    //If the sensor is not connected disable the timer and interrupt
    else if(digitalRead(DetectPin1) == LOW){

        timerWindSpeed.stop();
        detachInterrupt(WindspeedPin);
        WindsensorConnected=0;
        WindsensorState="Disconnected";
        RevCount=0;
        Speed=0;
        DirectionNumber=0;
        //Serial.println("Windsensor is not connected");
        PowerSave1=1;
    }
}

```

```

    }

    strcpy(Direction, DirectionDescription[DirectionNumber]); //Retrieves
    wind direction measurement from array
    //Serial.printlnf("Wind direction is: %s", Direction);

}

//Setting up the timer and initialisation of the interrupt responsible for
wind sensor measurements
void WindsensorSetup(){

    Speed=0;
    RevCount=0;
    timerWindSpeed.reset(); //The stopped timer will reset and with the
    next code it will start
    timerWindSpeed.start(); //Initialize timer for wind speed calculation
    attachInterrupt(WindspeedPin, Revolution, FALLING); //Setup an
    interrupt that is triggered on the falling edge of the signal every
    time the reed switch closes
    WindsensorConnected=1;
    PowerSave1=0; //Switch sleep mode off

}

//Interrupt driven subroutine that executes when a pulse is detected
void Revolution(){

    //Software debounce
    if(millis()-DebounceTime > 15){

        RevCount=RevCount+1;
        DebounceTime=millis();

    }

}

//Executed every 4 seconds to obtain the wind speed

```

```

void WindSpeed(){

    Speed=(3.621015*RevCount)/(SampleTime1/1000); //Speed in km/hr
    RevCount=0;
    timerWindSpeed.reset(); //Resets the timer to start from 0

}

//Obtaining wind direction
void WindDirection(){

    RawValue = analogRead(WindDirectionPin);
    DirectionNumber = map(RawValue, 0, 4096, 1, 9);
    //Serial.println("Wind direction analog voltage: %fV", AnalogValue);

}

```

Code Listing 4.3: Wind Speed and Wind Direction Measurements

The formula 4-1, adapted from the Davis 6410 datasheet, is used to calculate the wind speed (“Wind Speed Translation Formula”) [11]:

$$WindSpeed = \frac{P(2.25)(1.60934)}{T_{sample}} \quad [4-1]$$

where the number of interrupts (P) that occur within the 4 second sample time (T_{sample}), multiplied by a factor (2.25) and a conversion factor (1.60934) to convert the wind speed from miles per hour to km/hr [38]. The use of a 4 second sample time affords the wind speed sensor a resolution of 0.905 km/hr which is acceptable for this design.

The software debounce time was determined by considering:

- The maximum wind speed required by this design;
- The typical amount of time taken by a switch to settle after experiencing a change of state.

Since the maximum wind speed to be measured by this sensor is known to be 190 km/hr, the number of interrupt triggers per second (f) was calculated, using equation 4-1, to be 53. With

the frequency known, the period (T) was calculated to be 18 ms using the formula $T = \frac{1}{f}$ [39]. Therefore, the software debounce time could not exceed 18 ms or pulses from the sensor will be missed at wind speeds approaching 190 km/hr. After some research into typical switch debounce times [40], a debounce time of 15 ms was deemed to be suitable for this application.

The wind direction is determined using particle's mapping function [41]: since the maximum voltage measurable is 3.3 V and 8 points of cardinal direction is required, each direction has a voltage band of 0.4125 V. Wind direction measurements are only considered in the presence of non-zero wind speed conditions. This is to prevent false wind direction readings under no-wind conditions.

4.5. Rain Sensor Operation and Programming

It consists of the primary subroutine "RainSensor()" and 3 secondary subroutines, shown in **Code Listing 4.4**, that are responsible for:

- Initialising and terminating the interrupt responsible for tracking the raw wind rain data;
- Enabling and disabling of the timer that is used to calculate rain measurements;
- Determining the connection status of the rain sensor;
- Obtaining the rain sensor measurements in mm/hr.

```
//Primary rain sensor subroutine
void RainSensor(){

    if(digitalRead(RainDetectorPin)==HIGH){

        if(RainSensorConnected==0){

            RainSensorSetup();
            //Serial.println("Rain sensor is connected and setup is
            complete");
            RainSensorState="Connected";

        }

    }

}
```

```

//If the rain sensor is not connected disable the interrupt and timer
else{

    detachInterrupt(RainSensorPin);
    timer2.stop();
    TipCount=0;
    RainFall=0;
    RainSensorConnected=0;
    RainSensorState="Disconnected";
    //Serial.println("Rain sensor is not connected");
    PowerSave2=1;

}

}

//Setting up the timer and initialisation of the interrupt responsible for
rain sensor measurements
void RainSensorSetup(){

    TipCount=0;
    attachInterrupt(RainSensorPin, Tipping, RISING);
    timer2.reset();
    timer2.start();
    RainSensorConnected=1;
    PowerSave2=0;

}

//Interrupt driven subroutine that executes when a pulse is detected
void Tipping(){

    //Software debounce
    if(millis()-DebounceTime1 >15){

        TipCount=TipCount+1;
        DebounceTime1=millis();

    }

}

```

```

//Executes every 15 minutes to obtain rainfall measurements
void Rain(){

    RainFall=((TipCount*0.2)*(3600000/SampleTime2)); //Rainfall in mm/hr
    TipCount=0;
    timer2.reset();
    timer2.start();

}

```

Code Listing 4.4: Rain Sensor Operation

The number of interrupts (T_{pulses}) that each represents 0.2 mm of rain, which occur within the 15-minute sample time (t_{sample}), is used to calculate the rainfall rate ($RainFall$) in millimetres per hour (mm/hr), according to the formula developed from the equation in the BGT WS-601ABS2 datasheet 4-2 [15]:

$$RainFall = (T_{pulses} \times 0.2) \times \frac{60}{t_{sample}} \quad [4-2]$$

The use of the 15-minute sample time affords the rain sensor a resolution of 0.8 mm/hr which is acceptable for this design. Since the frequency of reed switch action for this sensor is lower than the reed switch action of the wind speed sensor, the same debounce time was used to minimise the firmware cycle period and for the purpose of standardisation.

4.6. Power Saving Features

Two power saving features have been implemented for this design:

- Since the weather station is not required to communicate via Wi-Fi, the P0 Wi-Fi module on the Photon was switched off using “SYSTEM_MODE(MANUAL)” [42];
- The second power saving feature involved placing the Photon in a sleep mode (“Stop mode”) where execution of code stops until either an external source triggers an interrupt or a predetermined period has elapsed [43]. This feature was implemented, after the Li-ion battery was selected and procured, to extend the station’s battery life by reducing its power requirements. There were 2 challenges associated with this mode that needed to be addressed before it became viable:

- This feature is only activated, as shown in **Code Listing 4.5** when both the wind and rain sensors are not connected to the weather station as pulses from either of these sensors will be ignored when the Photon is asleep, leading to false measurements;

```
//Checks if sleep mode should be implemented
void PowerSaveMode(){

    if(PowerSave1==1 & PowerSave2==1){
        //Power saving mode activated when both the rain sensor
        and the wind sensor are disconnected from the station

        System.sleep(DetectPin1,RISING,SleepTime);
        CheckDataLoggerPB();
        //Update the state of the data logger PB after
        returning from stop mode

    }
}
```

Code Listing 4.5: Controlling the Execution of Sleep Mode

- The duration of the sleep mode had to be minimised to 1 second in order to allow the weather station to maintain a stable connection to the Blynk mobile app.

4.7. Mobile App Operation and Programming

The “BlynkComms()” subroutine, shown in **Code Listing 4.6**, is responsible for the communication between the weather station’s hardware and the app. This subroutine is called last in order to allow the other subroutines to first update the data stored in their variables before it is transmitted. Each variable is allocated to a widget that appears on the app, using Blynk’s virtual pins, which allows the data to be transmitted and displayed easily.

```
//Responsible for sending weather, sensor connection status and battery
status data to the Blynk mobile application
void BlynkComms(){
```



```

//BME280 sensor data
Blynk.virtualWrite(V0, Temperature);
Blynk.virtualWrite(V1, Humidity);
Blynk.virtualWrite(V2, Pressure);
Blynk.virtualWrite(V8, BME280SensorState); //Connection status

//Wind sensor data
Blynk.virtualWrite(V5, Direction);
Blynk.virtualWrite(V6, Speed);
Blynk.virtualWrite(V9, WindsensorState); //Connection status

//Rainfall sensor data
Blynk.virtualWrite(V15, RainFall);
Blynk.virtualWrite(V16, RainSensorState); //Connection status

//Battery monitoring subroutine
BatteryStatus();
}

```

Code Listing 4.6: Communication with the Blynk App

4.7.1 The Blynk Security Feature

Although the initialisation command “Serial1.begin(115200)” enables the weather station hardware to communicate via the JDY08 BLE module, a unique token key generated by Blynk must be included in the weather station’s firmware to allow communication between the hardware and the app [44]. This prevents other unauthorised applications, created on the Blynk platform, from accessing the weather station.

4.8. The Operation and Programming of the Battery Status Monitoring Feature

The “BatteryStatus()” subroutine, shown in **Code Listing 4.7**, is responsible for:

- Determining the battery voltage as well as the “Battery normal”, “Battery low”, “Battery charging” and “Battery charged” statuses;
- Updating the battery voltage and status widgets on the weather station app;
- Triggering app notifications to indicate battery statuses.

```

void BatteryStatus(){

    BatteryVoltage=((analogRead(BatteryPin)*4.2)/3510));
    //Serial.println("Battery voltage: %f V", BatteryVoltage);
    Blynk.virtualWrite(V13, BatteryVoltage);

    //Non-charging battery conditions
    if((digitalRead(STAT1)==HIGH && digitalRead(STAT2)==HIGH) ||
    (digitalRead(STAT1)==LOW && digitalRead(STAT2)==LOW)){

        Blynk.setProperty(V13, "label", "Battery voltage"); //Label
        battery voltage display widget

        BatteryChargedNotification=1; //Used to control the execution of
        the battery charged notification
        BatteryChargingNotification=1; //Used to control the execution of
        the battery charging notification

        //Check battery voltage
        if(BatteryVoltage<=3.65){ //Battery critically low voltage
            condition

            Blynk.setProperty(V12, "color", "#F00606"); //Red LED
            Blynk.setProperty(V12, "label", "Battery low"); //Set the
            widget to display that the battery is low

            //Serial.println("Battery critically low. Please connect
            charger immediately!");
            Blynk.notify("Battery critically low. Please connect charger
            immediately!");
        }

        //Battery low voltage condition
        else if(BatteryVoltage<=3.7 && BatteryLowNotification==1){

            Blynk.setProperty(V12, "color", "#F00606"); //Red LED
            Blynk.setProperty(V12, "label", "Battery low"); //Set the
            widget to display that the battery is low

```

```

        //Serial.println("Battery is low. Charge battery");
        Blynk.notify("Battery is Low. Please connect charger!");
        BatteryLowNotification=0;

    }

    //Normal operating Battery voltage condition
    else if(BatteryVoltage>3.7){

        Blynk.setProperty(V12, "color", "#23C48E");    //Green LED
        Blynk.setProperty(V12, "label", "Battery normal");    //Set the
        widget to display that the battery is normal
        //Serial.println("Battery normal");

    }

}

//Battery charging condition
if(digitalRead(STAT1)==LOW && digitalRead(STAT2)==HIGH){

    BatteryVoltageComparison=5;    //To ensure that this variable will
    be set with the new battery voltage under normal battery operating
    conditions
    BatteryChargedNotification=1;
    BatteryLowNotification=1;

    //Display the battery charging notification on the mobile app
    if(BatteryChargingNotification==1){

        Blynk.notify("Battery charging");
        BatteryChargingNotification=0;

    }

    Blynk.setProperty(V12, "color", "#DFED00");    //Yellow LED
    Blynk.setProperty(V12, "label", "Battery charging");    //Set the
    widget to display that the battery is charging
    Blynk.setProperty(V13, "label", "Battery charge voltage");

}

}

```

```

// Battery charged condition
if((digitalRead(STAT1)==HIGH && digitalRead(STAT2)==LOW)){

    Blynk.setProperty(V12, "color", "#1A6BF4");    //Blue LED
    Blynk.setProperty(V12, "label", "Battery charged");    //Set the
    widget to state that the battery is charged
    Blynk.setProperty(V13, "label", "Battery voltage");
    //Serial.println("Battery is Charged");

    BatteryChargingNotification=1;

    // Display the battery charged notification on the mobile app
    if(BatteryChargedNotification==1){

        Blynk.notify("Battery charged");
        BatteryChargedNotification=0;

    }

}
}

```

Code Listing 4.7: Determining the Battery Voltage and Status

The battery voltage is determined using equation 4-3:

$$V_{bat} = \frac{V_{raw} \times 4.2}{3510} \quad [4-3]$$

where V_{bat} is the battery voltage, V_{raw} is the digital equivalent of the analogue voltage across the 68 k Ω resistor that is measured by the ADC . This equation was developed by considering the following:

- Since the Photon's ADC has a 12-bit resolution, a binary value of 4095 represents its maximum measurable voltage of 3.3 V [8];
- The maximum battery voltage that is scaled down across the 68 k Ω resistor, which was determined by Ohm's law, is 2.828 V.

Since 3.3 V represents 4095, by proportion, the maximum binary value representing 2.828 V is 3510. Finally, the scaled down voltage measurements are scaled up by a multiplying factor of 4.2, the battery's maximum voltage, to achieve the measurement of the battery's actual voltage.

4.9. Data Logging Operation and Programming

The firmware segment required to operate the data logger consists of the "BLYNK_WRITE(V14)" subroutine, which is executed when the data logger push button on the app is toggled, and the "SaveToSDCard()" subroutine. The "SaveToSDCard()" subroutine is programmed, as shown in **Code Listing 4.8**, to:

- Generate a timestamp from the Photon's RTC and log timestamped weather data to the micro SD card when the data logger feature is activated and a micro SD card is present;
- Deactivate the data logger when no SD is present or when the SD card is removed while data logging is still active;
- Generate notifications on the mobile interface to indicate the change of state of the data logger.

```
//Executes when the data logger PB is toggled on the app
BLYNK_WRITE(V14){

    DataLogger = param.asInt();    //The state of the app PB

    //Activate data logging
    if(DataLogger==HIGH){

        CheckSDCard=1;

    }
    else{

        Blynk.notify("Save data to SD card: Disabled");
        //Serial.println("Save data to SD card: Disabled");

    }
}
```

```

//Primary subroutine that enables/disables data logging to the SD card
void SaveToSDCard(){

    //Execute when the data logger feature is enabled through the app
    if(SDActive==1){

        //Checks if the SD card is connected
        if (digitalRead(DetectPin2)==HIGH){

            //Seperate the data batches stored on the SD card
            if(CheckSDCard==1){

                printToCard.println("\nNew batch of weather
                data,Date,Time,Temperature(C),Humidity(%),Pressure(hPa),
                Wind speed(km/hr),Wind direction,Light intensity(klx),
                Rainfall(mm/hr)"); //This will run once only when the SD
                is first activated
                //Serial.println("\nNew batch of weather data");

                Blynk.notify("Data logger: Enabled");
                //Serial.println("Data logger: Enabled");
                CheckSDCard=0;

            }

            //Log timestamped data to SD card
            else if(CheckSDCard==0){

                //Obtain current date and time
                t=Time.now();
                Seconds=Time.second(t);
                Minutes=Time.minute(t);
                Hours=Time.hour(t);
                Day=Time.day(t);
                Month=Time.month(t);
                Year=Time.year(t);

                printToCard.printf(" ,%i/%i/%i,%i:%i:%i,%0.2f,%0.2f,

```

```

        %0.2f,%0.2f,%s,%0.2f,%0.2f", Day, Month, Year, Hours,
        Minutes, Seconds, Temperature, Humidity, Pressure, Speed,
        Direction, RainFall);
    //Serial.println("%i/%i/%i %i:%i:%i Temperature: %0.2f
    C, Humidity: %0.2f %, Pressure: %0.2f hPa, Wind speed:
    %0.2f km/hr, Wind direction: %s, Rainfall: %0.2f mm", Day,
    Month, Year, Hours, Minutes, Seconds, Temperature,
    Humidity, Pressure, Speed, Direction, RainFall);

    }
}

//Executes if the data logging is active/activated but no SD card
is present in the SD card reader
else{

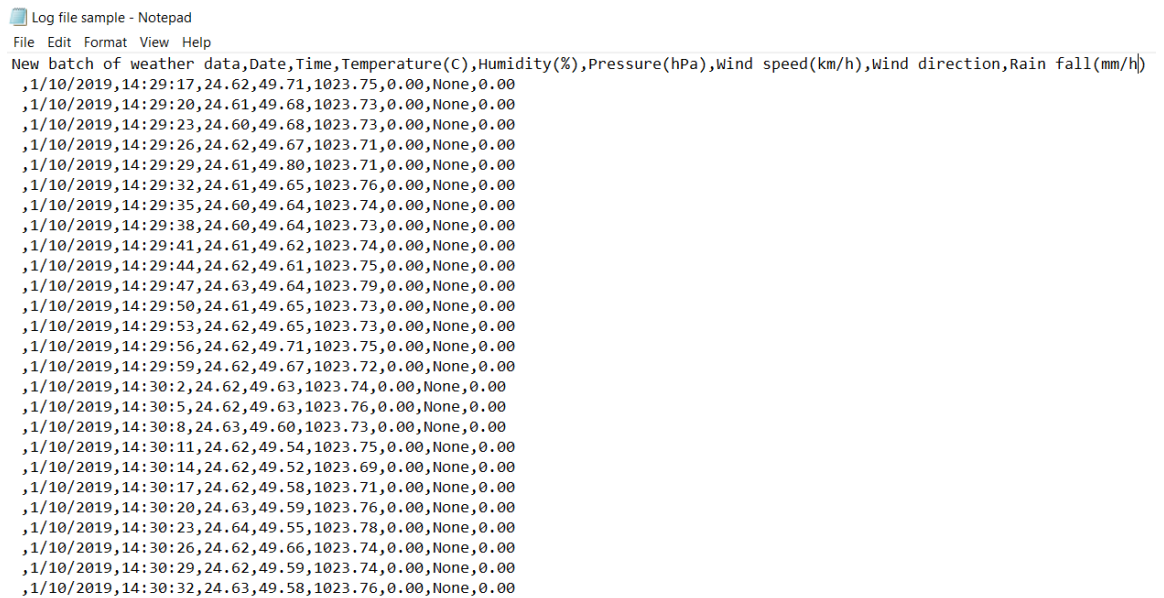
    //Serial.println("No SD card present");
    Blynk.notify("No SD card present");
    Blynk.virtualWrite(V14,0);
    SDOption=0;
    SDActive=0;
    counter2=0;

    }
}
}

```

Code Listing 4.8: Data Logger Operation

The data is stored as a text file on the SD card periodically, every 3 seconds, in a comma delimited format which allows the logged data to be imported easily into Microsoft Excel for data manipulation and trend generation [45]. A sample of the log file is shown in **Figure 4.1**.



```

Log file sample - Notepad
File Edit Format View Help
New batch of weather data,Date,Time,Temperature(C),Humidity(%),Pressure(hPa),Wind speed(km/h),Wind direction,Rain fall(mm/h)
,1/10/2019,14:29:17,24.62,49.71,1023.75,0.00,None,0.00
,1/10/2019,14:29:20,24.61,49.68,1023.73,0.00,None,0.00
,1/10/2019,14:29:23,24.60,49.68,1023.73,0.00,None,0.00
,1/10/2019,14:29:26,24.62,49.67,1023.71,0.00,None,0.00
,1/10/2019,14:29:29,24.61,49.80,1023.71,0.00,None,0.00
,1/10/2019,14:29:32,24.61,49.65,1023.76,0.00,None,0.00
,1/10/2019,14:29:35,24.60,49.64,1023.74,0.00,None,0.00
,1/10/2019,14:29:38,24.60,49.64,1023.73,0.00,None,0.00
,1/10/2019,14:29:41,24.61,49.62,1023.74,0.00,None,0.00
,1/10/2019,14:29:44,24.62,49.61,1023.75,0.00,None,0.00
,1/10/2019,14:29:47,24.63,49.64,1023.79,0.00,None,0.00
,1/10/2019,14:29:50,24.61,49.65,1023.73,0.00,None,0.00
,1/10/2019,14:29:53,24.62,49.65,1023.73,0.00,None,0.00
,1/10/2019,14:29:56,24.62,49.71,1023.75,0.00,None,0.00
,1/10/2019,14:29:59,24.62,49.67,1023.72,0.00,None,0.00
,1/10/2019,14:30:02,24.62,49.63,1023.74,0.00,None,0.00
,1/10/2019,14:30:05,24.62,49.63,1023.76,0.00,None,0.00
,1/10/2019,14:30:08,24.63,49.60,1023.73,0.00,None,0.00
,1/10/2019,14:30:11,24.62,49.54,1023.75,0.00,None,0.00
,1/10/2019,14:30:14,24.62,49.52,1023.69,0.00,None,0.00
,1/10/2019,14:30:17,24.62,49.58,1023.71,0.00,None,0.00
,1/10/2019,14:30:20,24.63,49.59,1023.76,0.00,None,0.00
,1/10/2019,14:30:23,24.64,49.55,1023.78,0.00,None,0.00
,1/10/2019,14:30:26,24.62,49.66,1023.74,0.00,None,0.00
,1/10/2019,14:30:29,24.62,49.59,1023.74,0.00,None,0.00
,1/10/2019,14:30:32,24.63,49.58,1023.76,0.00,None,0.00

```

Figure 4.1: Image Showing a Sample of the Logged Weather Data on the SD Card

4.10. Problems Encountered and their Solutions

4.10.1 Obtaining and Maintaining Time on the Photon

One of the requirements of the data logging feature is to timestamp the data entries that are saved to the micro SD card. The first challenge posed by this requirement was to obtain the time and date information even though the weather station does not have access to Wi-Fi and therefore, particle's cloud servers. However, since the weather station connects to the mobile app, which is connected to Blynk's servers via an internet connection, the weather station is able to acquire the Unix time from Blynk's servers using the RTC widget [46]. The Unix time is then converted to the standard date and time format using particle's predefined functions [47]. The second challenge that needed to be addressed was maintaining the time offline: since access to Blynk's server is only possible through the connection to the mobile app, the time cannot be maintained by Blynk when the weather station is disconnected from the app. To resolve this, each time that the weather station hardware is connected to the app for the first time (which will always be necessary to initiate operation of the weather station), the Unix time is obtained and is used to set Photon's RTC. The Photon's RTC then maintains the time offline. The segment of firmware responsible for this is shown in **Code Listing 4.9:**

```
//Executes on start-up of the weather station
```



```

void setup() {

    //...Initialisations...

    //Subroutine called to obtain Unix time
    GetTime();

}

//Requests the Unix time from Blynk
void GetTime(){

    requestTime();

}

//This subroutine is returned when the Blynk Unix time is requested
void requestTime(){

    Blynk.sendInternal("rtc", "sync");

}

//This subroutine is executed when the Blynk Unix time is returned
BLYNK_WRITE(InternalPinRTC) {

    //The Unix time returned by Blynk
    t = param.asLong();

    //Unix time obtained from Blynk sets the Photon's RTC
    Time.setTime(t);

}

```

Code Listing 4.9: Obtaining and Maintaining Time

4.10.2 Fault Finding and Debugging

During the developmental stages of the firmware, problems would arise as the firmware was altered and refined to build better functionality into it. It was not clear, in some instances, if the cause of the problem was in the firmware or a result of damaged hardware. The initial individual programs that were developed were found to be useful in this case as they provided an effective means of isolating functionality and establishing the problem area. The use of the serial monitor proved to be a valuable debugging tool in cases where a program compiled successfully but did not function correctly. Although Particle's online dashboard serves a similar purpose, the use of a serial monitor was favoured since this project was designed to be operated offline. The programming statements responsible for displaying data to the serial monitor was intentionally commented out and left in the firmware so that it can be used for fault finding purposes during the lifespan of the weather station.

CHAPTER 5: HARDWARE TESTING AND RESULTS

This chapter details the experiments that were implemented to test each hardware element of this design and the analysis of the results that were obtained.

5.1 Davis Wind Speed Sensor

5.1.1 Wind Speed Sensor Testing

The wind speed sensor was first tested for consistency using a two-speed household hair dryer that was able to produce wind at a consistent and repeatable speed. A consistent wind speed measurement was a strong indicator that the sensor and the programming code were functioning correctly. Once this was established, a RS212-578 wind meter was used as a standard to test the accuracy of the Davis' wind speed measurements. The Davis sensor and the standard were mounted, next to each other, and centred in front an ordinary household 3-Speed fan which was used as the wind source. The diagram of this is shown in **Figure 5.1**.

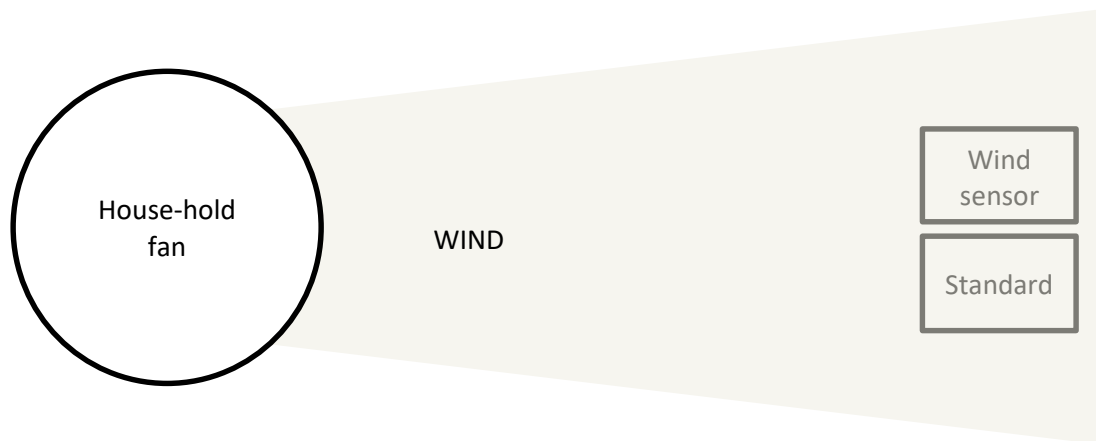


Figure 5.1: Adopted Method for Wind Speed Sensor Testing

To simulate different wind conditions:

1. The fan's rotational speed was adjusted by cycling through its 3 speed settings;
2. The distance between the fan and the wind speed sensor/meter was varied.

5.1.2 Wind Speed Sensor Test Results

Wind speeds were only able to be produced up to a maximum of 22 km/h, which is far below the required 190 km/h. In order to gauge the performance of the wind speed sensor, the correlation between the wind speed measurements produced by the sensor and the RS212-578 wind meter standard needed to be established first. These measurements are shown in **Table 5.1**.

Sample	RS212-578 wind speed meter standard (km/hr)	Davis wind speed sensor (km/hr)
0	0	0
1	2,6	2,7
2	3,5	3,6
3	4,2	4,5
4	4,7	5,4
5	6,1	6,3
6	6,5	7,2
7	6,6	7,3
8	9,9	10,8
9	11,8	12,6
10	12,1	12,7
11	15,7	16,3
12	17,5	18,1
13	19,7	20,6
14	20,4	21,2
15	22,2	23,1

Table 5.1: Wind Speed Measurements Produced by the Wind Speed Sensor and the Wind Speed Meter

Since the correlation co-efficient was calculated to be 0.9996, the response of the sensor was verified to be linear and the equation that describes the curve of best fit was found to be:

$$y_{WindSensor} = 1.032x_{Standard} + 0.224 \quad [5-1]$$

where $y_{WindSensor}$ represents the Davis wind speed measurements and $x_{Standard}$ represents the wind speed measurements of the RS212-578 wind speed meter. Therefore, the wind speed sensor was concluded to be a linear device and no linearization was required in the firmware. The measured values, including the line of best fit (described by equation 5-1), is shown in **Figure 5.2**.

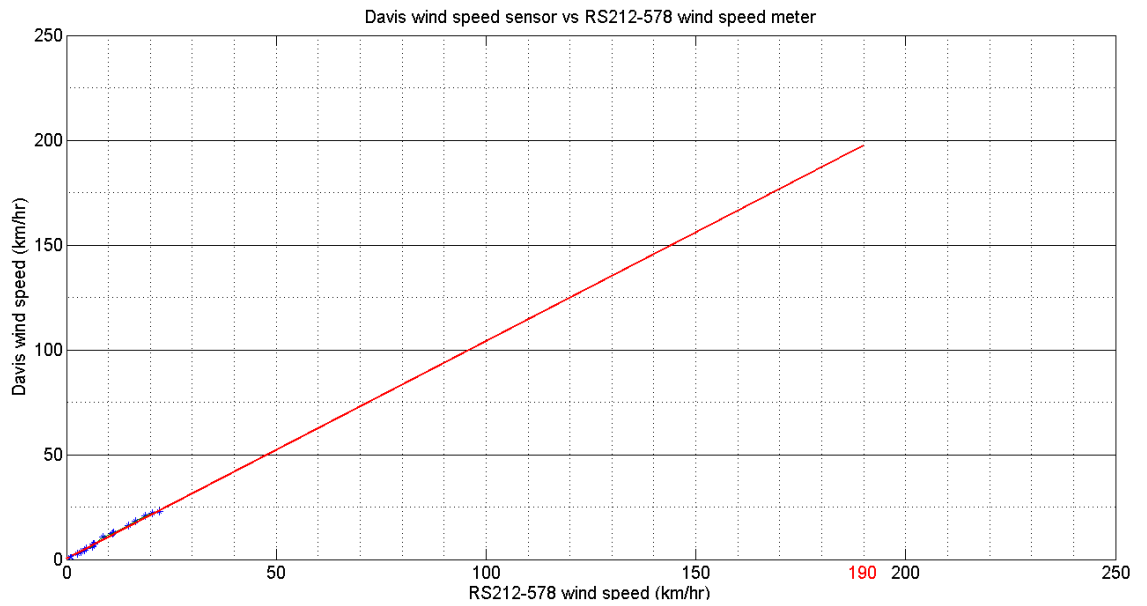


Figure 5.2: Scatter Graph Showing the Correlation of Wind Speed Measurements between the Davis Sensor and the RS212-578 Wind Speed Meter

The wind speeds measured by the Davis wind sensor and RS212-578 standard were compared and the results are shown in **Figure 5.3**:

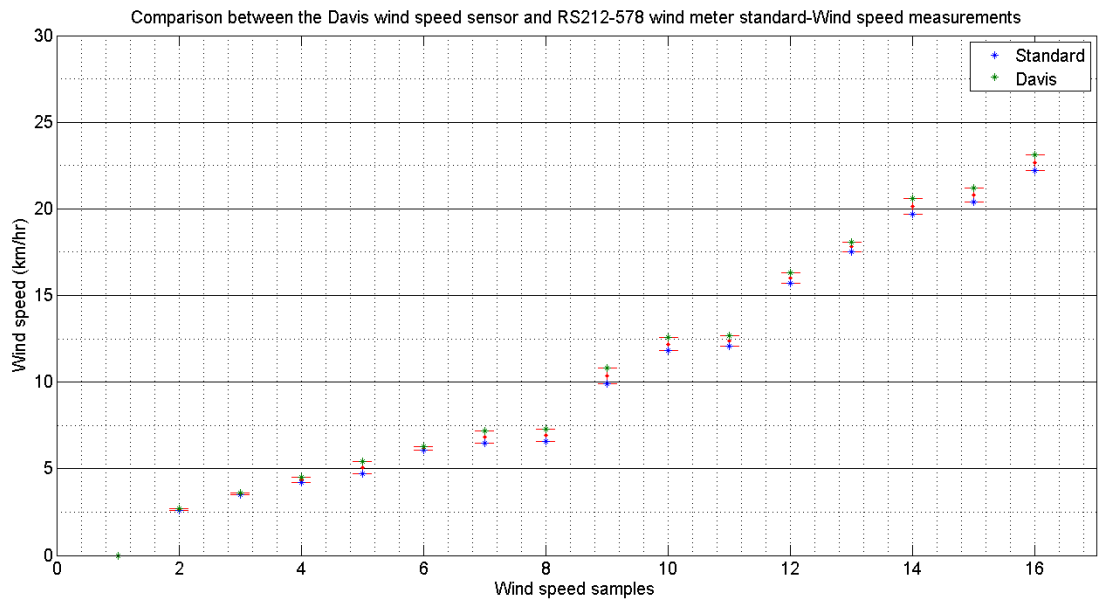


Figure 5.3: Comparison between the Wind Speeds Measured by the Davis Wind Speed Sensor and RS212-578 Wind Speed Meter

The maximum deviation between the sensor and the standard was 8.3% (0.9 km/h), which is within the 10% tolerance specified for this design. Therefore, the accuracy of this sensor was deemed to be adequate and no calibration was required.

5.2 Davis Wind Direction Sensor

5.2.1 Wind Direction Sensor Testing

The wind direction sensor was set and tested as follows:

- A template was created, using a mechanical compass and protractor, that consisted of a circle that was divided into 16 equal parts (22.5° apart): 8 equal parts (45° apart) to represent the directions required by this design as well as the 8 points in between each direction showing the exact degree and voltage at which the direction indication transitions from one indication to the other. The circle was then further divided into 5° intervals and a smaller circle, with a radius equal to the base of the direction sensor (10.5 mm), was cut out of the centre of the circle as indicated in

Figure 5.4:

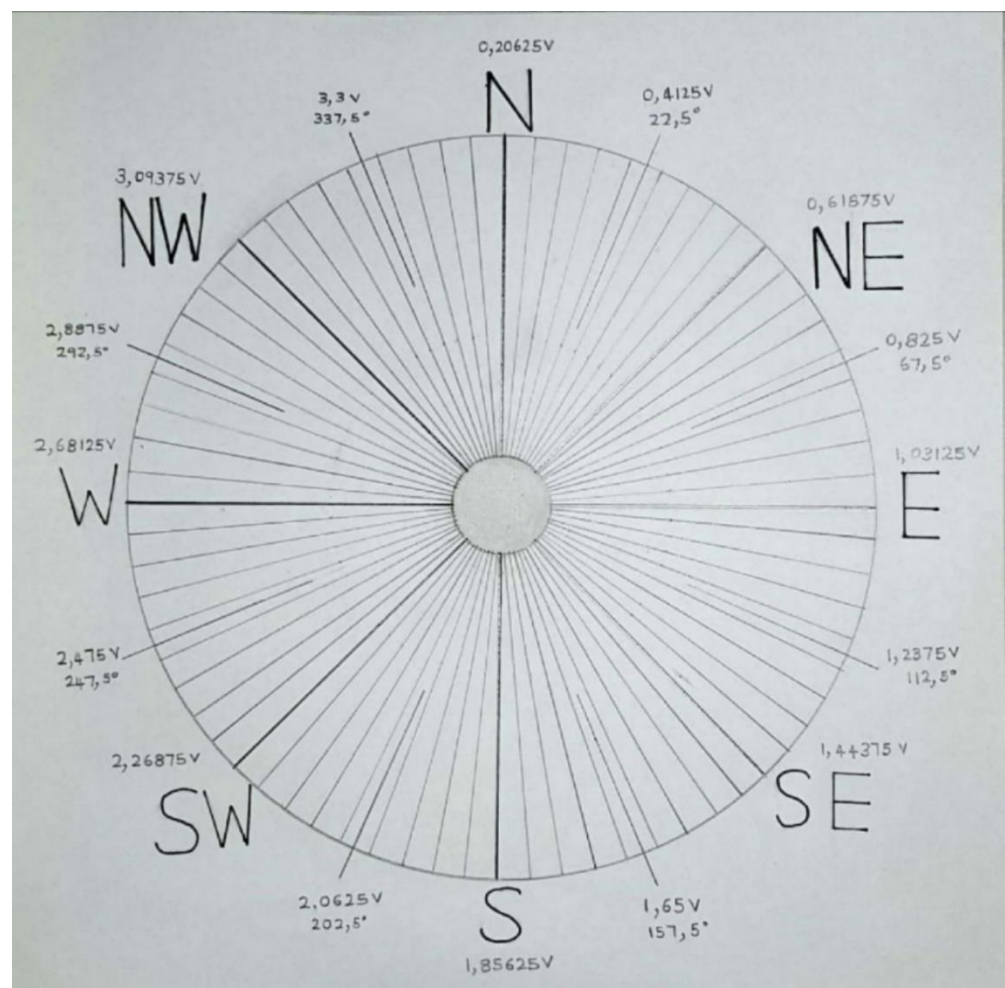


Figure 5.4: Template Used to Set the Wind Direction Sensor

- The wind sensor was mounted, the sensor's arrow head was removed and the template was fixed to the base of the sensor;
- The potentiometer's output voltage was adjusted to 0.20625 V (the centre point of the North direction's 0.4125 V voltage band) and was observed on the serial monitor using the code segment shown in **Code Listing 5.1**:

```
void WindDirection() {

    //Obtain voltage of wind sensor potentiometer and the
    //corresponding direction indication
    RawValue = analogRead(AnalogPin1);
    DirectionNumber = map(RawValue, 0, 4096, 1, 9);

    //Formula to calculate the voltage measurement
    AnalogValue = (3.3 * RawValue) / 4095;

    strcpy(Direction, DirectionDescription[DirectionNumber]);
    Serial.printlnf(" Pot voltage %fV, Direction %s",
        AnalogValue, Direction);

}
```

Code Listing 5.1: Setting the Direction Sensor

The formula shown in the code segment of **Code Listing 5.1** was developed using the same methodology mentioned in section 4.8 that was used to develop equation 4-3;

- The arrow head of the wind sensor was then reconnected and secured tightly, to the potentiometer's shaft, in the position pointing completely toward the North (N) direction drawn on the page as shown in **Figure 5.5**:

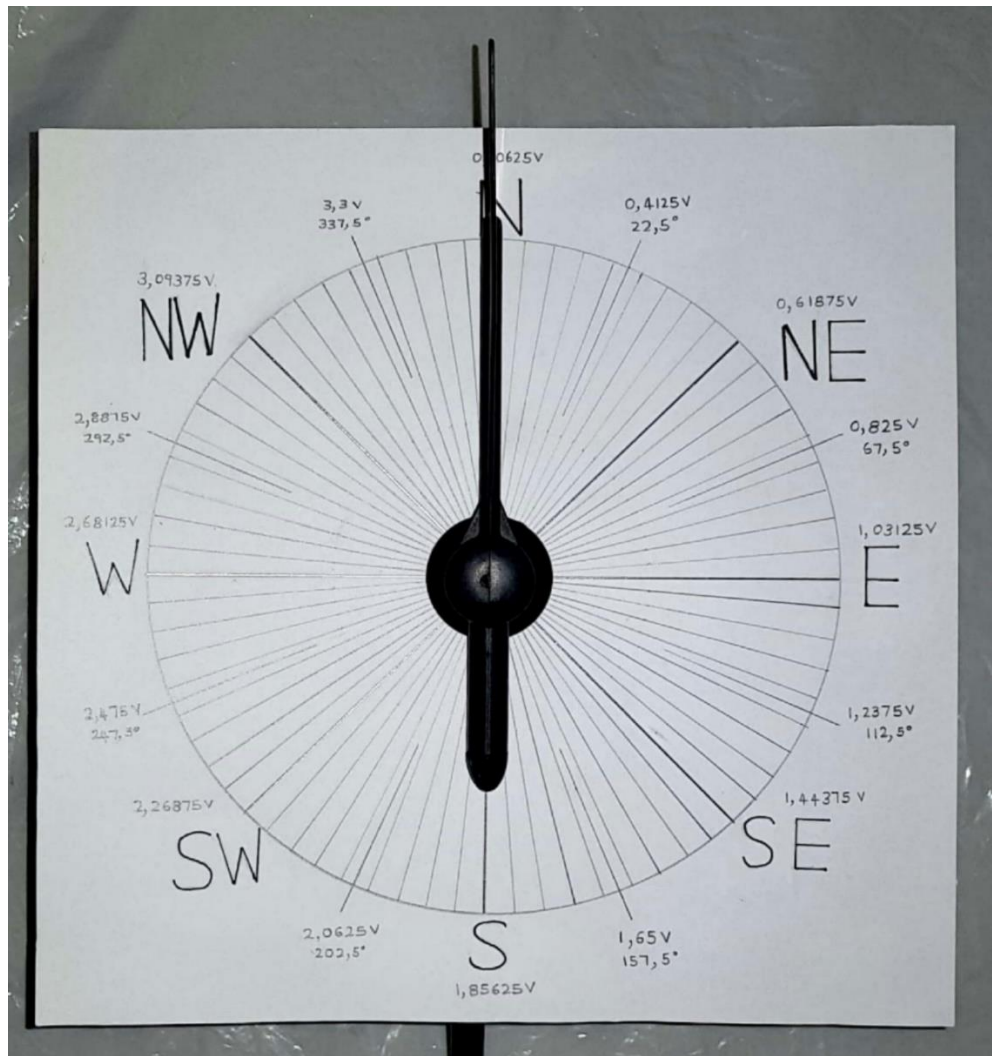


Figure 5.5: Setting of the Wind Direction Sensor Towards North

The wind vane was rotated and the direction indicated by the sensor at the 8 Cardinal and 8 transition points were observed on the serial monitor and compared to the direction points indicated on the template.

5.2.2 Wind Direction Sensor Test Results

The maximum degree of deviation between the sensor's direction indication and the direction template was 4°. Since this is within the 5° tolerance required by the design specification, the sensor's accuracy was deemed to be acceptable for this project and therefore no calibration was required.

5.3 BME280 Temperature Sensor

5.3.1 Temperature Sensor Testing

The BME280's temperature measurements were tested against a Fluke 52 K/J thermometer standard. Since no laboratory grade equipment was available, the testing process involved the use of a household two-bar heater, a refrigerator and an ice bath. The two sensors were placed side by side and centred on a flat surface facing the heater. Their distance from the heater was varied in order to test their measuring ability at the higher end of the temperature range (24 °C to 55 °C). To test temperature conditions between 4 °C and 8 °C, the two sensors were placed inside a refrigerator and its temperature was varied. To obtain temperatures close to 0 °C, the sensors were placed inside a re-sealable plastic bag which was submerged in an ice bath. Care was taken to ensure that no water leaked into the partially opened top end of the bag, to prevent water damage to the sensors. Readings were taken every 5 minutes, until stable readings were reached, to accommodate the slow thermal response of the sensors.

5.3.2 Temperature Sensor Test Results

The testing method described in section 5.3.1, although reliable, was only able to produce temperature samples that ranged from 1.2 °C to 55.1 °C which is short of the required -22 °C to 55 °C temperature range. The correlation between the temperature measurements produced by the BME280 temperature sensor and the Fluke 52 thermometer was investigated in order to gauge the performance of the temperature sensor. The readings produced by both devices is shown in **Table 5.2**.

Sample	Fluke 52 thermometer standard (°C)	BME280 Temperature sensor (°C)
1	0,3	1,2
2	3,1	4,4
3	5,7	6,8
4	24,1	25,5
5	28,6	29,9
6	29,5	30,6
7	33,8	34,2
8	36,9	37,2
9	37,7	37,7
10	38,5	38,5

11	40,2	41,1
12	44,3	45,1
13	48,6	49,4
14	52,5	53,6
15	55,1	56,2

Table 5.2: Temperature Measurements Produced by the BME280 Temperature Sensor and the Fluke 52 Thermometer

Since the correlation co-efficient was calculated to be 0.9997, the response of the sensor was verified to be linear and the equation that describes the curve of best fit was found to be:

$$y_{TempSensor} = 0.992x_{Standard} + 1.072 \quad [5-2]$$

where $y_{TempSensor}$ is the resulting BME280 temperature measurements and $x_{Standard}$ is the temperature measurements of the Fluke 52 thermometer. Therefore, the temperature sensor was concluded to be a linear device and no linearization was required in the firmware. The measured values, including the line of best fit, described by equation 5-2, is shown in **Figure 5.6**

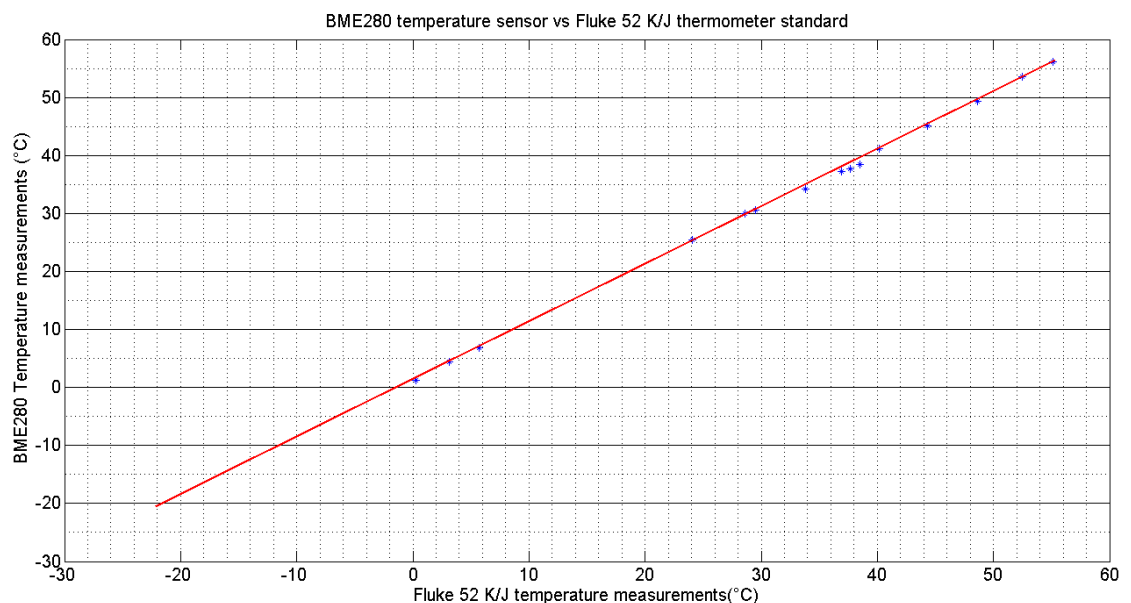


Figure 5.6: Scatter Graph Showing the Correlation of Temperature Measurements between the BME280 Temperature Sensor and the Fluke 52 Thermometer

The temperature measurements produced by the BME280 sensor and the Fluke 52 thermometer were compared and the results are shown in **Figure 5.7**

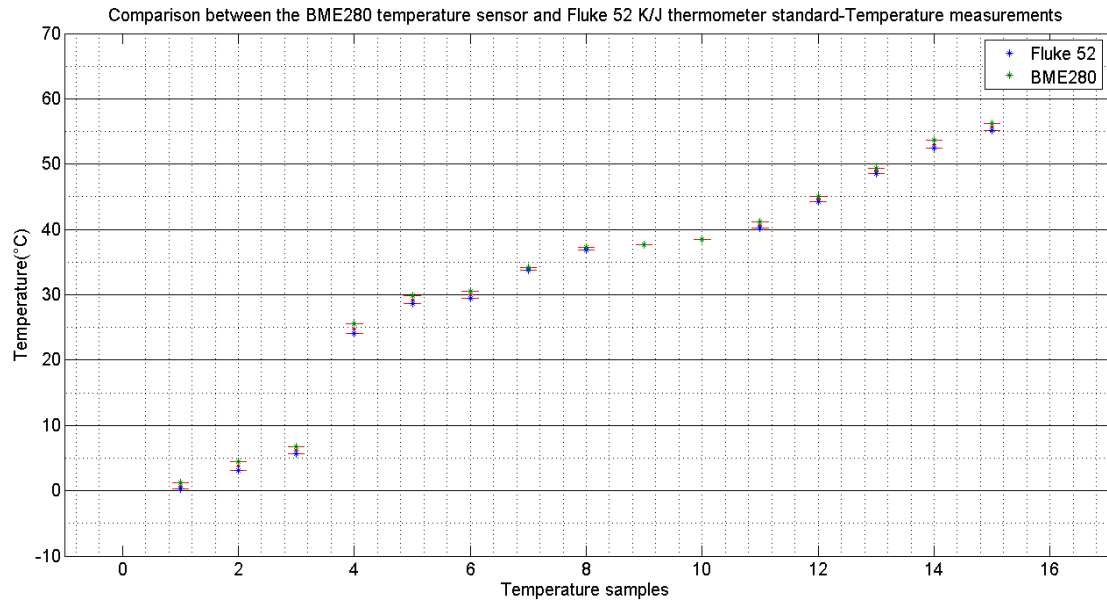


Figure 5.7: Comparison Between the Temperature Measurements Produced by the BME280 Sensor and the Fluke 52 Thermometer

The BME280 temperature sensor produced an acceptable maximum deviation of 1.4 °C from the Fluke 52 thermometer. Since this is within the 2 °C tolerance specified for this design, no calibration was required.

5.4 BME280 Humidity Sensor

5.4.1. Humidity Sensor Testing

Since no calibrated instrument standard was available to test the accuracy of the BME280 humidity sensor, this sensor's measurements were compared to the humidity readings produced by the weather station standard installed at Virginia Airport. The BME280's humidity sensor was setup in the vicinity of Virginia Airport's weather station and measurements were compared in 15 minute intervals over a 4-hour period.

5.4.2. Humidity Sensor Test Results

Access to Virginia Airport's weather station was limited and humidity readings produced during that time frame only ranged from 51% to 80% which was short of the required 0% to 100% humidity range. In order to gauge the performance of the humidity sensor, the

correlation between the humidity measurements produced by the sensor and the weather station standard at Virginia Airport needed to be established first. The readings produced by both devices are shown in **Table 5.3**.

Sample	Weather station standard at Virginia Airport (%)	BME280 humidity sensor (%)
1	51	50,3
2	53	52,1
3	58	57,1
4	61	60,3
5	63	62,5
6	67	66
7	70	70,6
8	72	73,1
9	73	74,3
10	74	75,1
11	76	75,9
12	77	76,7
13	78	78,2
14	80	80,2

Table 5.3: Humidity Measurements Produced by the BME280 Sensor and the Weather Station Standard

Since the correlation co-efficient was calculated to be 0.9980, the response of the sensor was verified to be linear and the equation that describes the curve of best fit was found to be:

$$y_{HumSensor} = 1.053x_{Standard} - 3.711 \quad [5-3]$$

where $y_{HumSensor}$ is the resulting BME280 humidity measurements and $x_{Standard}$ is the humidity measurements of Virginia Airport's weather station standard. Therefore, the humidity sensor was concluded to be a linear device and no linearization was required in the firmware. The measured values, including the line of best fit (described by equation 5-3), is shown in **Figure 5.8**.

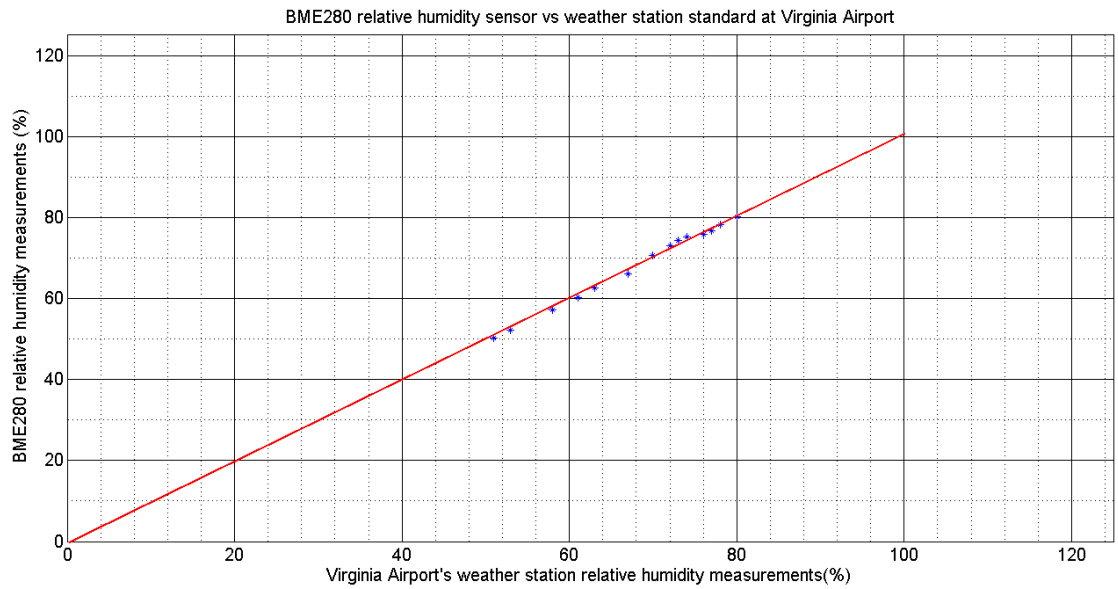


Figure 5.8: Scatter Graph Showing the Correlation of Measurements Produced by the BME280 Humidity Sensor and the Weather Station Standard

The humidity measurements produced by the BME280 sensor and the weather station standard were compared and the results are shown in **Figure 5.9**

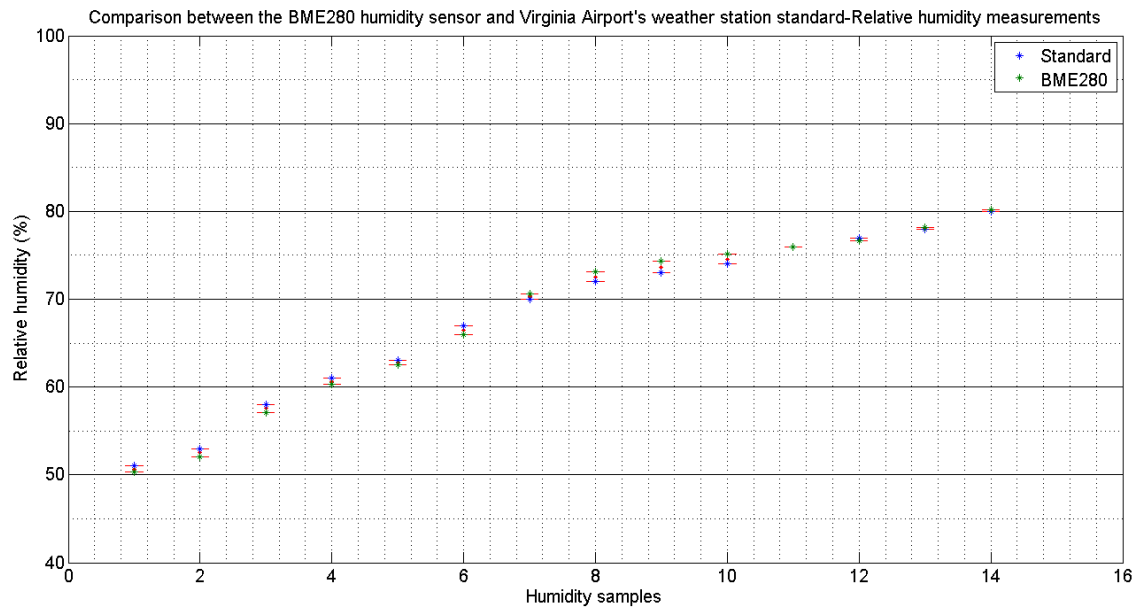


Figure 5.9: Comparison Between the Humidity Measurements Produced by the BME280 Sensor and the Weather Station Standard

The maximum deviation was found to be 1.75% which is within the 5% tolerance required by the design specifications. Therefore, no calibration was required.

5.5 BME280 Pressure Sensor

5.5.1. Pressure Sensor Testing

Since no pressure meter standard was available to test the accuracy of the BME280's pressure sensor and the weather station installed at Virginia Airport had a pressure sensor that produced measurements from a constant altitude (sea-level), a different approach was required. A method that involved the use of equation 5-4 [48] was used to verify the pressure readings produced by the sensor at various locations, around KwaZulu-Natal, that were different altitudes above sea level [49]:

$$p = p_0 \times \left(1 - \frac{Lh}{T_0}\right)^{\frac{gM}{RL}} \quad [5-4]$$

where p is the pressure in Pa, p_0 is the pressure at sea level in Pa, L is the temperature lapse rate 0.0065 K/m, h is the altitude in meters above sea level, T_0 is the atmospheric temperature in Kelvin, g is earth-surface gravitational acceleration 9.80665 m/s², M is the molar mass of dry air 0.0289644 kg/mol and R is the universal gas constant 8.31447 J/(mol.K). This equation was favoured over the use of equation 1-1 because it includes the effect that the temperature has on the atmospheric pressure. Since the atmospheric temperature was measured by the BME280's temperature sensor at each location, a more accurate pressure measurement was able to be calculated using equation 5-4.

5.5.2. Pressure Sensor Test Results

The maximum accessible altitude of 659 m (at location 29°46'36.3"S 30°45'51.8"E) and the pressure at sea level on the day that testing was conducted, limited the pressure samples to be between 934.5 hPa and 1011 hPa. This was short of the required 600 hPa to 1050 hPa pressure range. The correlation between the pressure measurements produced by the sensor and the calculated standard needed to be established first in order to gauge the performance of the sensor. The readings that were produced are shown in **Table 5.4**.

Sample	Calculated standard (hPa)	BME280 pressure sensor (hPa)
1	934,54	935,1

2	936,5	936,96
3	938,58	938,83
4	949,86	950,23
5	960,096	960,56
6	977,82	978,41
7	996,47	996,8
8	997,62	998,1
9	998,66	999,1
10	999,35	999,8
11	999,92	1000,324
12	1004,08	1003,651
13	1005,24	1004,766
14	1007,09	1006,879
15	1008,4	1008,03
16	1009,6	1009,305
17	1011	1011

Table 5.4: Pressure Measurements Produced by the BME280 Sensor and the Weather Station Standard

Since the correlation co-efficient was calculated to be 0.9999, the response of the sensor was verified to be linear and the equation that describes the curve of best fit was found to be:

$$y_{PressSensor} = 0.992x_{standard} + 7.716 \quad [5-5]$$

where $y_{PressSensor}$ is the resulting BME280 pressure measurements and $x_{standard}$ is the pressure measurements of the calculated standard. Therefore, the pressure sensor was concluded to be a linear device and no linearization was required in the firmware. The measured values, including the line of best fit (described by equation 5-5), is shown in **Figure 5.10**.

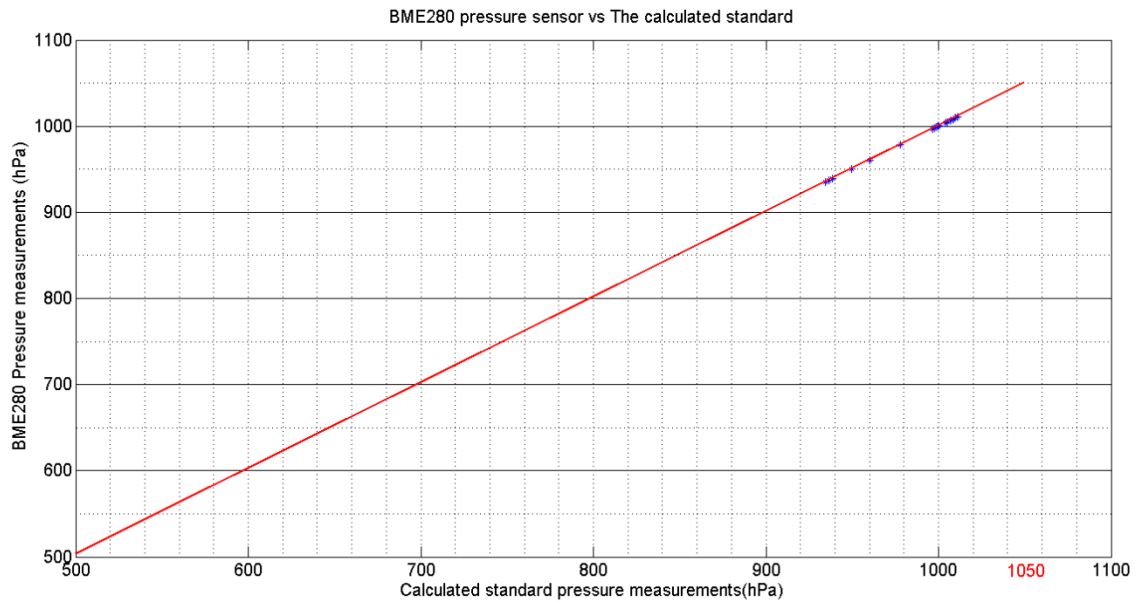


Figure 5.10: Scatter Graph Showing the Correlation of Measurements Produced by the BME280 Pressure Sensor and the Calculated Standard

The pressure measurements produced by the BME280 sensor and the calculated standard were compared and the results are shown in **Figure 5.11**

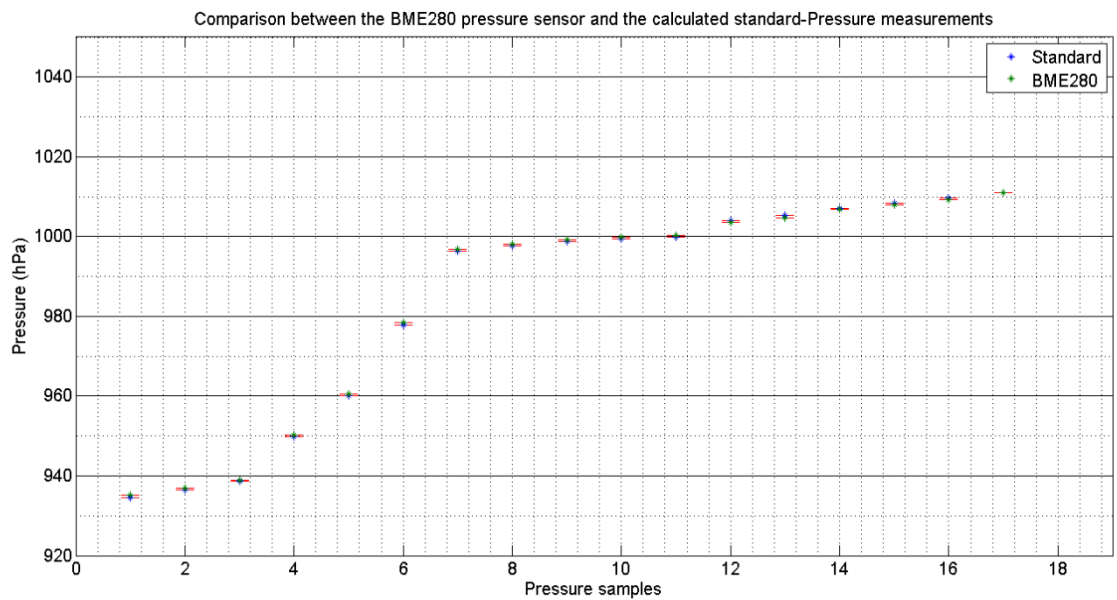


Figure 5.11: Comparison Between the Humidity Measurements Produced by the BME280 Sensor and the Weather Station Standard

The BME280's pressure sensor produced a maximum deviation of 1.46 hPa from the calculated standard. Since this is within the 10 hPa tolerance specified for this design, no calibration was required.

5.6 BGT WS-601ABS2 Rain Sensor

5.6.1 Rain Sensor Testing

The rain sensor was tested by checking the correspondence between the rain measurements that should have resulted from the known volumes of water that were poured slowly (to simulate rain) into the tipping bucket according to the equation 4-2, and the actual rain measurements that were produced by the sensor. It was ensured that the rain sensor was placed level so that a representative result was reflected. The smaller volumes of water, used to test the sensor, were obtained using a syringe while a measuring cylinder was used to obtain the larger volumes of water that measured up to a maximum of 219.45 ml (which is equivalent to 28 mm/hr of rainfall). The water was poured in increments of 6.27 ml for the first 2 measurements, thereafter the increments were increased to 18.81 ml.

5.6.2 Rain Sensor Test Results

The rain sensor was tested over the full operating range of 0 to 28 mm/hr. **Table 5.5** shows the rain measurements produced by the rain sensor and the calculated standard.

Sample	Calculated standard (hPa)	BME280 pressure sensor (hPa)
1	934,54	935,1
2	936,5	936,96
3	938,58	938,83
4	949,86	950,23
5	960,096	960,56
6	977,82	978,41
7	996,47	996,8
8	997,62	998,1
9	998,66	999,1
10	999,35	999,8
11	999,92	1000,324
12	1004,08	1003,651
13	1005,24	1004,766
14	1007,09	1006,879
15	1008,4	1008,03
16	1009,6	1009,305
17	1011	1011

Table 5.5: Rain Measurements Produced by the Rain Sensor and the Calculated Standard

Since the correlation co-efficient was calculated to be 0.9995, the response of the sensor was verified to be linear and the equation that describes the curve of best fit was found to be:

$$y_{RainSensor} = 0.994x_{Standard} - 0.068 \quad [5-6]$$

where $y_{RainSensor}$ is the resulting BGT WS-601ABS2 rain measurements and $x_{Standard}$ is the rain measurements of the calculated standard. Therefore, the pressure sensor was concluded to be a linear device and no linearization was required in the firmware. The measured values, including the line of best fit (described by equation 5-6), is shown in **Figure 5.12**.



Figure 5.12: Scatter Graph Showing the Correlation of Measurements Produced by the BGT WS-601ABS2 Rain Sensor and the Calculated Standard

The rain measurements produced by the BGT WS-601ABS2 sensor and the calculated standard were compared and the results are shown in in **Figure 5.13**.

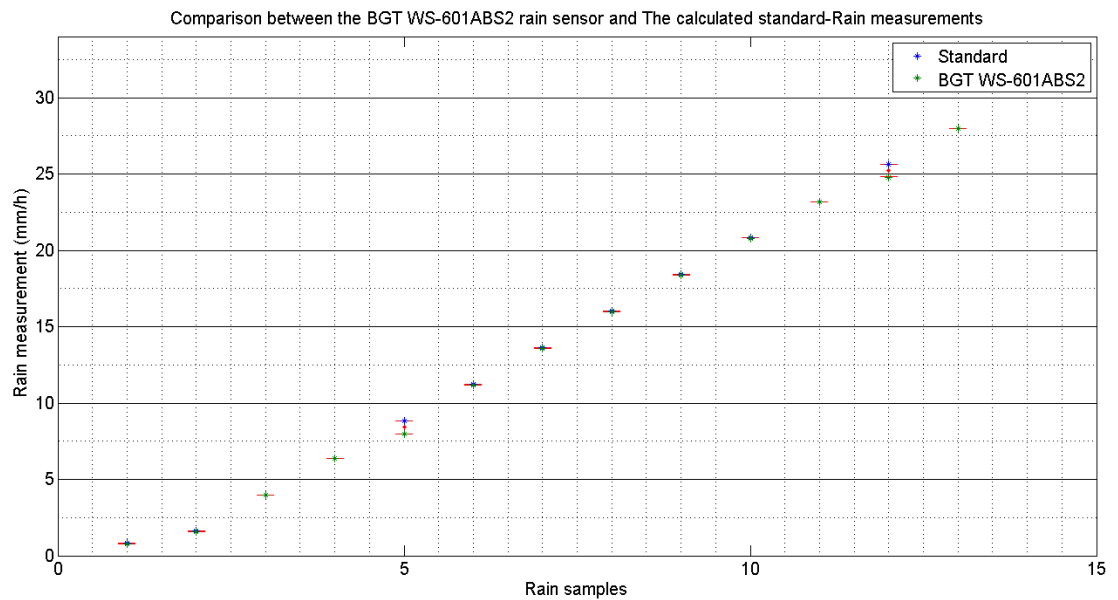


Figure 5.13: Comparison Between the Rain Measurements Produced by the Rain Sensor and the Calculated Standard

The maximum deviation produced by this rain sensor was 0.8 mm/hr from the calculated standard, which is within the design's 2 mm/hr tolerance. Therefore, no calibration was required.

5.7 Sensor Connection Testing and Results

The sensors' connection statuses were tested by connecting and disconnecting all the sensors to/from the weather station. The connection status for each sensor was tested using the segment of code shown in **Code Listing 5.2**.

```
//Prints the sensor's connection status to the serial monitor
Serial.print("\nTemperature, pressure and humidity sensor are ");
Serial.print(BME280SensorState);
Serial.println("");
Serial.print("Wind speed and direction sensors are ");
Serial.print(WindsensorState);
Serial.println("");
Serial.print("Rain fall sensor is ");
Serial.print(RainSensorState);
Serial.println("");
```

5.8 Power Consumption

5.8.1 Current Consumption Testing

The current consumption of the weather station was measured, using a Fluke 177 multimeter, after all the components of the weather station were connected. The current consumption was measured with:

1. No power management features implemented;
2. The Photon's Wi-Fi module "Off" (power saving feature 1);
3. The Photon's Wi-Fi module "Off" and the wind rain and wind sensors disconnected from the weather station so that sleep mode could be activated (power saving feature 1 and 2).

Since power saving feature 2 requires that the weather station cycle "ON" and "OFF" sleep mode, the current consumption that resulted in each state had to be measured separately so that the combined total current consumption could be determined. The length of the "OFF" state was increased to allow the multimeter sufficient time to make a stable measurement. The two measurements were then applied to the formula 5-7 to calculate the total current consumption effectively achieved by the weather station.

$$I_T = \left(I_{sleep} \times \frac{T_{sleep}}{T_{cycle}} \right) + \left(I_{awake} \times \frac{T_{awake}}{T_{cycle}} \right) \quad [5-7]$$

where I_T is the total effective average current (mA) consumed by the station, I_{sleep} is the current consumed (mA) when the Photon is in sleep mode, T_{sleep} is the amount of time (s) that the Photon is in sleep mode, I_{awake} is the current consumption (mA) when the Photon is not in sleep mode, T_{awake} (s) is the amount of time that the Photon is awake and T_{cycle} is the time taken for a single "On"/"Off" cycle.

5.8.2 Current Consumption Test Results

The results are summarised in the bar graph shown in **Figure 5.14**.

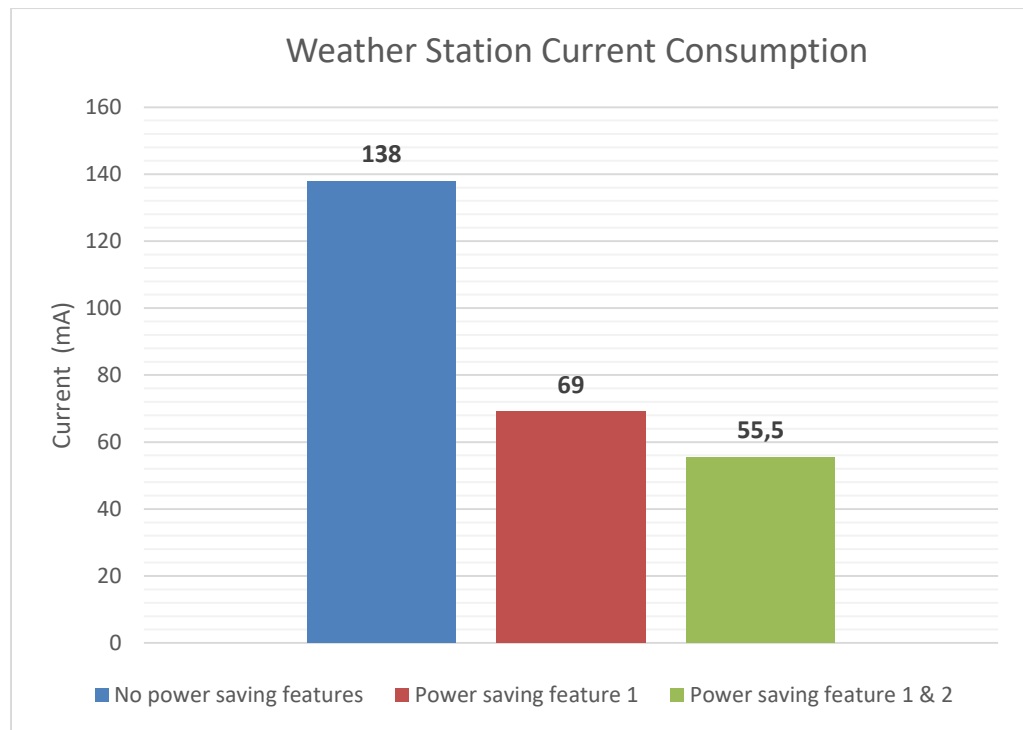


Figure 5.14: The Current Consumption of the Weather Station Before and After Implementing Power Saving Features

The implementation of the first power management feature reduced the station's current consumption from 137.6 mA to 69 mA. After implementing power management feature 2 the current consumption of the station was measured in both the "On" and "Off" states of sleep mode separately, to be 14.8 mA and 69 mA respectively, and then applying equation 5-7 yielded the effective average consumption to be 55.5 mA.

5.9 Data Logger

5.9.1 Data Logger Testing

The data logger was tested using 4 GB, 8 GB, 16 GB and 32 GB micro SD cards. This was to ensure the SD card reader's compatibility with SD cards up to 32 GB in size. The segment of code shown in **Code Listing 5.3**.

```

void SaveToSDCard(){

    //Writing to SD card activated
    if(digitalRead(D7)==HIGH){

        //Checks if the SD card is connected
        if (digitalRead(DetectPin2)==HIGH){

            //This will run when the SD is first activated
            if(Heading==1){

                printToCard.println("\nNew batch of weather
                data,Date,Time,Temperature(C),Humidity(%),
                Pressure hPa), Windspeed(km/h),Wind direction,
                Rain fall(mm/hr)");

                Serial.println("\nNew batch of weather
                data,Date,Time,Temperature(C),Humidity(%),
                Pressure hPa), Windspeed(km/h),Wind direction,
                Rain fall(mm/hr)");
                Heading=0;

                Serial.println("Data logger: enabled");

            }

            printToCard.printf(",%0.2f,%0.2f,%0.2f,%0.2f,%s,%0.2f,
            %0.2f, %d", Day, Month, Year, Hours, Minutes, Seconds,
            Temperature, Humidity, Pressure, Speed, Direction,
            RainFall);

            Serial.printf(",%0.2f,%0.2f,%0.2f,%0.2f,%s,%0.2f,
            %0.2f, %d", Day, Month, Year, Hours, Minutes, Seconds,
            Temperature, Humidity, Pressure, Speed, Direction,
            RainFall);

        }

    }
}

```

```

        else{
            //SD card is not present or is removed whilst logging data

            Serial.println("No SD card present");
            Heading=1;

        }
    }
    else if(digitalRead(D7)==LOW){
        //When data logging is de-activated

        Serial.println("Data logger: disabled");
        Heading=1;

    }
}

```

Code Listing 5.3: Segment of Code used to Test the Operation of the Data Logger

The micro SD card was inserted into the SD card module and data logging was initiated by setting input pin D7 “High”. After the weather station indicated the data was logged, the micro SD card was removed from the station, inserted into a mobile phone and checked to verify that data was written to the SD card. The data logging feature was tested under the following conditions:

- Data logging was initiated when the SD card was:
 - Present in the SD card reader;
 - Not present in the SD card reader.
- While data logging was active the SD card was removed from the SD card reader;
- Data logging was activated, with the SD card present, and after 1 minute it was deactivated.

5.9.2 Data Logger Test Results

Initially, the weather station was found to successfully log weather data to the 4 GB, 8 GB and 32 GB micro SD cards but failed to log data to the first 16 GB micro SD card that was tested. It was observed while testing the 16 GB card that the weather station indicated data was being logged to the card but when the card was inserted into a mobile device, the card

was found to be corrupt. Although testing on a second 16 GB card proved successful, this incident led to the realisation that the weather station produced false data logging indications because it was unable to detect this fault condition. To resolve this, the code segment shown in **Code Listing 5.3** was improved with the code segment shown in **Code Listing 5.4**:

```
void SaveToSDCard(){

    if(digitalRead(D7)==HIGH){ //Activate data logging

        //Checks if the SD card is connected
        if (digitalRead(DetectPin2)==HIGH){

            //Seperate the data batches stored on the SD card
            if(CheckSDCard==1){

                //This will run once only when the SD is first
                activated
                printToCard.println("\nNew batch of weather
                Data,Time,
                Temperature(C),Humidity(%),Pressure(hPa),Windspeed
                (km/hr),Wind direction, Rain fall(mm/hr)");

                Serial.println("\nNew batch of weather
                Data,Time,Temperature(C)
                ,Humidity(%),Pressure(hPa),Windspeed (km/hr),Wind
                direction, Rain fall(mm/hr)");

                //Check if there are errors when writing to the SD
                card
                if(!printToCard.getLastBeginResult()){

                    //If there are errors writing to SD card
                    Serial.println("Data logger initializing, please
                    wait...");
                    CheckSDCard=1;
                }
            }
        }
    }
}
```



```

        SDcount++;

        if(SDcount==10){
            //After 10 attempts if writing to the SD card
            fails, then the data logger is de-activated
            automatically and the weather station
            indicates a fault condition

            Serial.println("Data logger failed");
            SDcount=0;

        }
    }
    else{

        Serial.println("Data logger: Enabled");
        CheckSDCard=0;
        SDcount=0;

    }
}
else if(CheckSDCard==0){
    //Weather data is logged to the SD card

    printToCard.printf(",%0.2f,%0.2f,%0.2f,%0.2f,%s,%0.2f",
    Temperature, Humidity, Pressure, Speed, Direction,
    RainFall);

    Serial.printf(",%0.2f,%0.2f,%0.2f,%0.2f,%s,%0.2f",
    Temperature, Humidity, Pressure, Speed, Direction,
    RainFall);

}
}
else{//SD card is not present or is removed while logging
    data

    Serial.printf("No SD card present");
}

```

```

        CheckSDCard=1;

        SDcount=0;

    }
}
else if(digitalRead(D7)==LOW){
    //When data logging is de-activated

    Serial.println("Data logger: disabled");
    Heading=1;

}
}

```

Code Listing 5.4: Improved Code Segment for Data logging and Micro SD Card Detection

It was found that the log file is 0.95 MB and since no more than 3 files are required to log weather data over 24 hours, the weather station can log 3 years and 10 months of weather data on a 4 GB micro SD card.

5.10 Timestamp

5.10.1 Timestamp Testing

First the timestamp generated was verified on the serial monitor to ensure the date and time was produced correctly. Thereafter the accuracy of the timestamp maintained by the Photon's RTC was tested using the segment of code shown in **Code Listing 5.5:**

```

void GetTime(){

    if(RTCReset==0 && RTCEndTime<=Time.now()){ //When the period has
    elapsed

        requestTime(); //Requests the Unix time from Blynk

    }
}

```

```

//Executes after the Photon's RTC is reset by Blynk
if(RTCReset==1 && RTCEndTime<=Time.now()){

    RTCStartTime=tstart;
    RTCEndTime=RTCStartTime+RTCResetTime;    //'RTCResetTime'
    determines the period (12, 24, 36 hours)

    //The time at which the period began
    Seconds=Time.second(RTCStartTime);
    Minutes=Time.minute(RTCStartTime);
    Hours=Time.hour(RTCStartTime);
    Day=Time.day(RTCStartTime);
    Month=Time.month(RTCStartTime);
    Year=Time.year(RTCStartTime);

    printToCard.println("RTC START TIME: %i/%i/%i  %i:%i:%i ",
    Day,
    Month, Year, Hours, Minutes, Seconds);
    //Serial.println("RTC START TIME: %i/%i/%i  %i:%i:%i ",
    Day, Month, Year, Hours, Minutes, Seconds);

    //The future time at which the period will have elapsed
    Seconds=Time.second(RTCEndTime);
    Minutes=Time.minute(RTCEndTime);
    Hours=Time.hour(RTCEndTime);
    Day=Time.day(RTCEndTime);
    Month=Time.month(RTCEndTime);
    Year=Time.year(RTCEndTime);

    printToCard.println("RTC END TIME: %i/%i/%i  %i:%i:%i ",
    Day, Month, Year, Hours, Minutes, Seconds);
    //Serial.println("RTC END TIME: %i/%i/%i  %i:%i:%i ", Day,
    Month, Year, Hours, Minutes, Seconds);

    RTCReset=0;

}
}

```

```

BLYNK_WRITE(InternalPinRTC) {    //This function is executed when the
Blynk Unix time is returned

    //Last time indicated by the photon's RTC converted to normal
    date and time format before it is updated by Blynk
    t=Time.now();
    Seconds=Time.second(t);
    Minutes=Time.minute(t);
    Hours=Time.hour(t);
    Day=Time.day(t);
    Month=Time.month(t);
    Year=Time.year(t);

    printToCard.printlnf("Last time indicated by Photon's RTC :
%i/%i/%i %i:%i:%i ", Day, Month, Year, Hours, Minutes, Seconds);
    Serial.printlnf("Last time indicated by Photon's RTC : %i/%i/%i
%i:%i:%i ", Day, Month, Year, Hours, Minutes, Seconds);

    t = param.asLong();          //The Unix time returned by Blynk
    tstart=t;                    //The start of the RTC period

    //This Unix time returned by Blynk converted to normal date and
    time format
    Time.setTime(t);
    Seconds=Time.second(t);
    Minutes=Time.minute(t);
    Hours=Time.hour(t);
    Day=Time.day(t);
    Month=Time.month(t);
    Year=Time.year(t);

    printToCard.printlnf("Time returned by Blynk: %i/%i/%i %i:%i:%i
", Day, Month, Year, Hours, Minutes, Seconds);
    //Serial.printlnf("Time returned by Blynk: %i/%i/%i %i:%i:%i
", Day, Month, Year, Hours, Minutes, Seconds);

    RTCReset=1;

```

```
}
```

Code Listing 5.5: Testing the Accuracy of the Photon's RTC

The timestamp was logged to the micro SD card over periods of 12, 24 and 36 hours. At the end of each period the Photon requested the current time and date from the Blynk app and logged the result. The timestamp data on the SD card was then analysed by comparing the time and date reflected by the Photon's RTC and the time and date indicated by Blynk.

5.10.2 Timestamp Test Results

Testing yielded the results shown in **Table 5.6**:

Period (hours)	Photon RTC time	Blynk server time	Drift (seconds)
12	06-05-2019 22:10:43	06-05-2019 22:10:42	1
24	06-05-2019 22:21:13	07-05-2019 22:21:11	2
36	09-05-2019 10:37:29	09-05-2019 10:37:26	3

Table 5.6: Photon's RTC Time Vs Blynk's RTC Time

The Photon's RTC time was observed to drift by 1 second every 12 hours which is not uncommon for the RTCs used in Particle's devices [50]. This was corrected in firmware as shown in **Code Listing 5.6**:

```
void GetTime(){

    if(RTCReset==0 && RTCEndTime<=Time.now()){

        requestTime(); //Requests the Unix time from Blynk

        //The Photon's RTC is corrected every 12 hours
        if(float(((Time.now()-RTCStartTime)/RTCTimeAdjust))>=
            RTCResetTime) {

            //Every 12 hours the photons RTC time is adjusted by 1
            second
        }
    }
}
```

```

        t=(Time.now()-AdjustmentAmount);
        Time.setTime(t);    //Set the photon's RTC time

        Seconds=Time.second(t);
        Minutes=Time.minute(t);
        Hours=Time.hour(t);
        Day=Time.day(t);
        Month=Time.month(t);
        Year=Time.year(t);

        printToCard.println("Photon RTC time was adjusted by %i at:
%i/%i/%i %i:%i:%i ", AdjustmentAmount, Day, Month, Year,
Hours, Minutes,
Seconds);

        Serial.println("Photon RTC time was adjusted by %i at:
%i/%i/%i
%i:%i:%i ", AdjustmentAmount, Day, Month, Year, Hours,
Minutes, Seconds);
        RTCTimeAdjust++;    //The number of 12-hour periods that
elapsed

    }
}

//Executes after the photon's RTC is reset by Blynk
if(RTCReset==1 && RTCEndTime<=Time.now()){

    RTCStartTime=tstart;
    RTCEndTime=RTCStartTime+RTCResetTime;    //The time at which
the photon's RTC needs to be reset by Blynk

    Seconds=Time.second(RTCStartTime);
    Minutes=Time.minute(RTCStartTime);
    Hours=Time.hour(RTCStartTime);
    Day=Time.day(RTCStartTime);
    Month=Time.month(RTCStartTime);

```

```

    Year=Time.year(RTCStartTime);

    printToCard.println("RTC START TIME: %i/%i/%i  %i:%i:%i ",
    Day, Month, Year, Hours, Minutes, Seconds);

    //Serial.println("RTC START TIME: %i/%i/%i  %i:%i:%i ",
    Day, Month, Year, Hours, Minutes, Seconds);

    Seconds=Time.second(RTCEndTime);
    Minutes=Time.minute(RTCEndTime);
    Hours=Time.hour(RTCEndTime);
    Day=Time.day(RTCEndTime);
    Month=Time.month(RTCEndTime);
    Year=Time.year(RTCEndTime);

    printToCard.println("RTC END TIME: %i/%i/%i  %i:%i:%i ",
    Day, Month, Year, Hours, Minutes, Seconds);

    //Serial.println("RTC END TIME: %i/%i/%i  %i:%i:%i ", Day,
    Month, Year, Hours, Minutes, Seconds);

    RTCReset=0;

}

}

BLYNK_WRITE(InternalPinRTC) {  //This function is executed when the Blynk
Unix time is returned

    //Last time indicated by the photon's RTC converted to normal
    date and time format before it is updated by blynk
    t=Time.now();
    Seconds=Time.second(t);
    Minutes=Time.minute(t);
    Hours=Time.hour(t);
    Day=Time.day(t);
    Month=Time.month(t);
    Year=Time.year(t);

```

```

printToCard.println("Last time indicated by Photon's RTC :
%i/%i/%i %i:%i:%i ", Day, Month, Year, Hours, Minutes, Seconds);

Serial.println("Last time indicated by Photon's RTC : %i/%i/%i
%i:%i:%i ", Day, Month, Year, Hours, Minutes, Seconds);

t = param.asLong(); //The Unix time returned by Blynk
tstart=t;           //The start of the RTC period

//This Unix time returned by Blynk converted to normal date and
time format
Time.setTime(t);
Seconds=Time.second(t);
Minutes=Time.minute(t);
Hours=Time.hour(t);
Day=Time.day(t);
Month=Time.month(t);
Year=Time.year(t);

printToCard.println("Time returned by Blynk: %i/%i/%i %i:%i:%i
",
Day, Month, Year, Hours, Minutes, Seconds);

Serial.println("Time returned by Blynk: %i/%i/%i %i:%i:%i ",
Day, Month, Year, Hours, Minutes, Seconds);

RTCReset=1;

}

```

Code Listing 5.6: Improving the Accuracy of the Photon's RTC

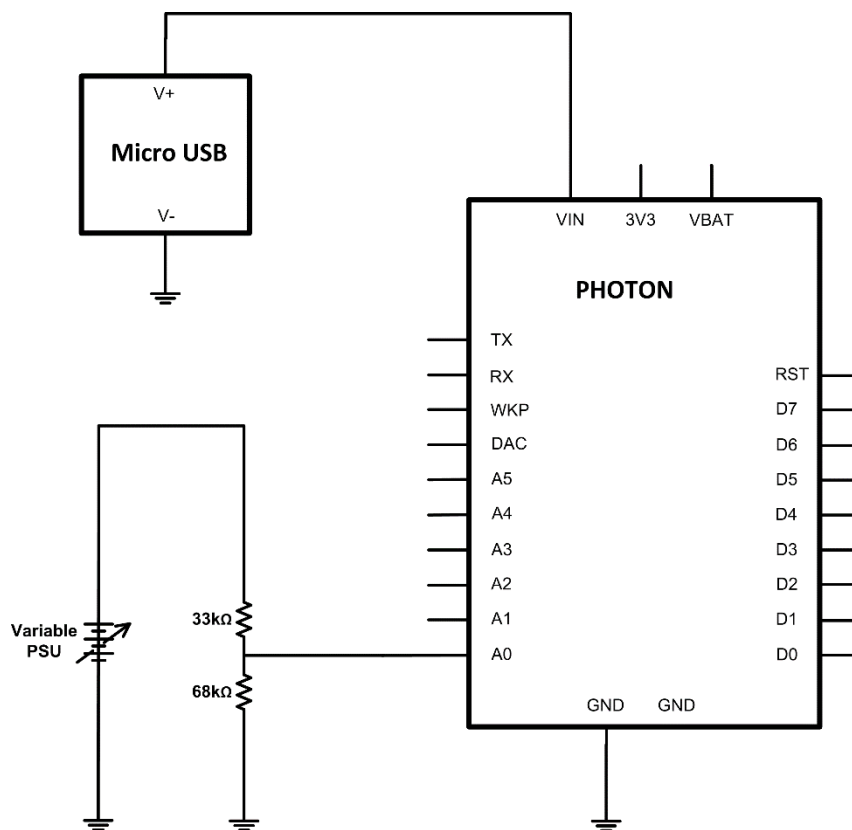
The Photon is programmed to reset its RTC with Blynk's server time, any time a connection to Blynk is available, after the 12-hour period has elapsed. Additionally, the Photon's RTC time is reduced by 1 second every 12 hours in the event that the weather station does not have access to Blynk after the 12-hour period has expired. The accuracy of the photon's RTC

was retested after implementing this method and it was found to have no observable drift from the time indicated by Blynk's servers.

5.11 Photon's Voltage Measurement

5.11.1 Voltage Measurement Testing

The accuracy of the voltage measurements produced by the Photon, on the ADC's channel 0, was tested using a variable bench power supply unit (PSU). The PSU was applied as an input to the battery monitoring circuitry as shown in **Figure 5.15**:



**Figure 5.15: Test Circuit used to Verify the Accuracy of the Voltage Measurements
Produced by the Photon**

A Fluke 177 multimeter standard was used to measure the output voltage of the PSU, as the PSU's voltage was varied, and its readings were compared to Photon's voltage measurements which were displayed on the serial monitor. Initially this test was conducted over the range of 3.6 V to 4.2 V, but since this range was changed (the reason for this is explained later in

section 6.5.1), the testing procedure was applied to the updated battery voltage range of 3.36 V to 4.2 V.

5.11.2 Voltage Measurement Test Result

The Photon’s measurements were tested over the 3.36 V to 4.2 V range, and the results are shown in **Figure 5.16**.

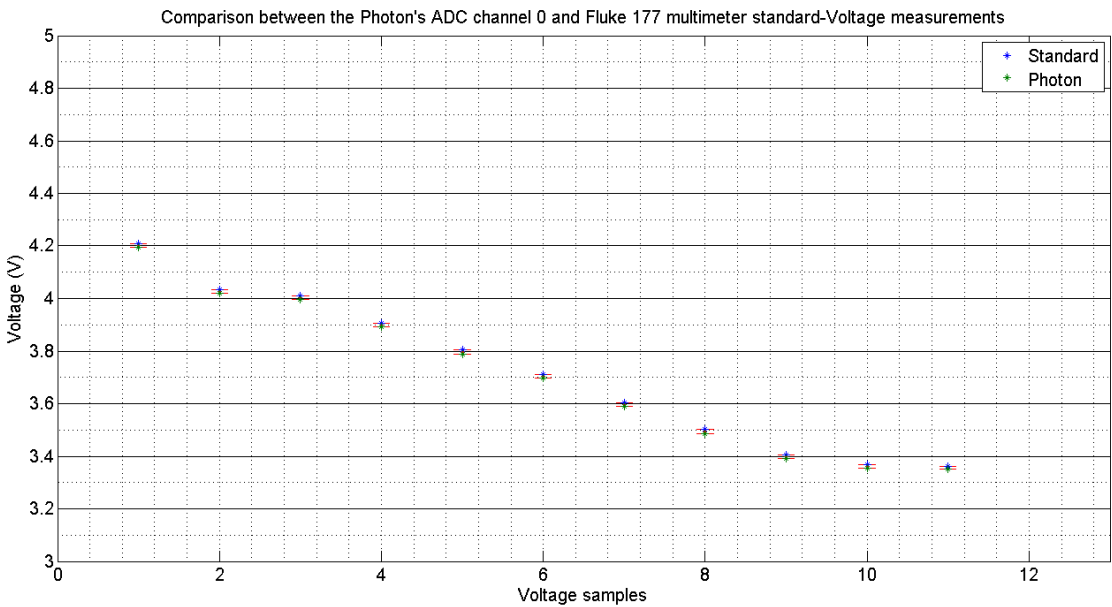


Figure 5.16: Comparison Between the Voltage Measurements Produced by the Photon and the Standard

The maximum, minimum and mean deviations produced were 0.017 V, 0.013 V and 0.015 V. Since this deviation was almost constant, the Photon’s voltage measurements were adjusted in the firmware by adding a factor of 0.015 to the result of equation 4-3. Then, the test procedure was repeated and the graph in **Figure 5.17** shows the calibrated results:

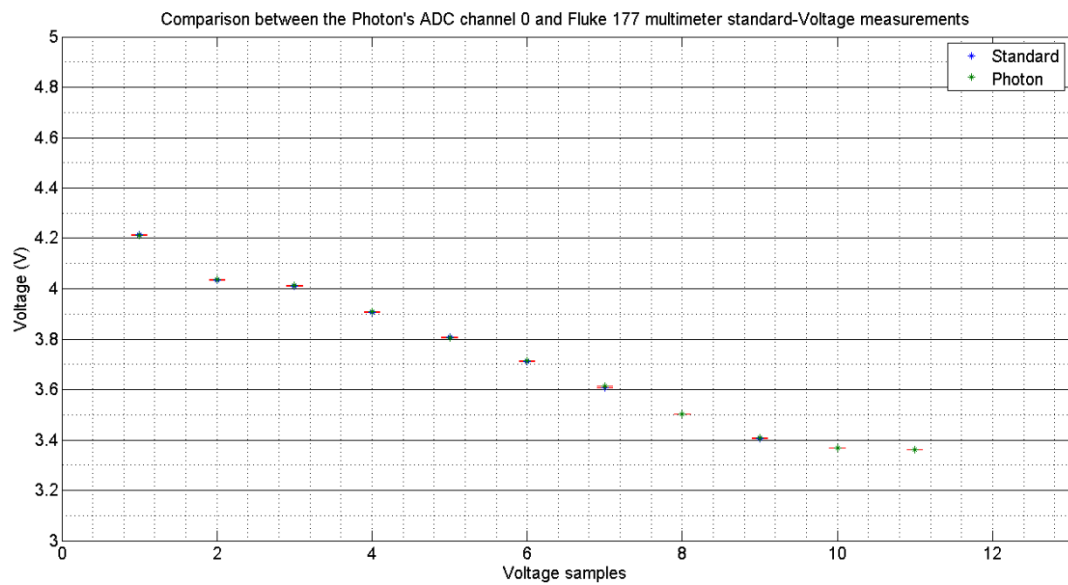


Figure 5.17: Comparison Between the Calibrated Voltage Measurements Produced by the Photon’s Channel 0 against the Fluke 177 Multimeter Standard

Since the maximum deviation produced after calibration was 0.009 V, which is less than 0.021%, it was deemed acceptable for the purposes of this design project.

5.12 Battery Status Monitoring

5.12.1 Battery Status Testing

The test procedure required the circuit configuration shown in **Figure 5.18**:

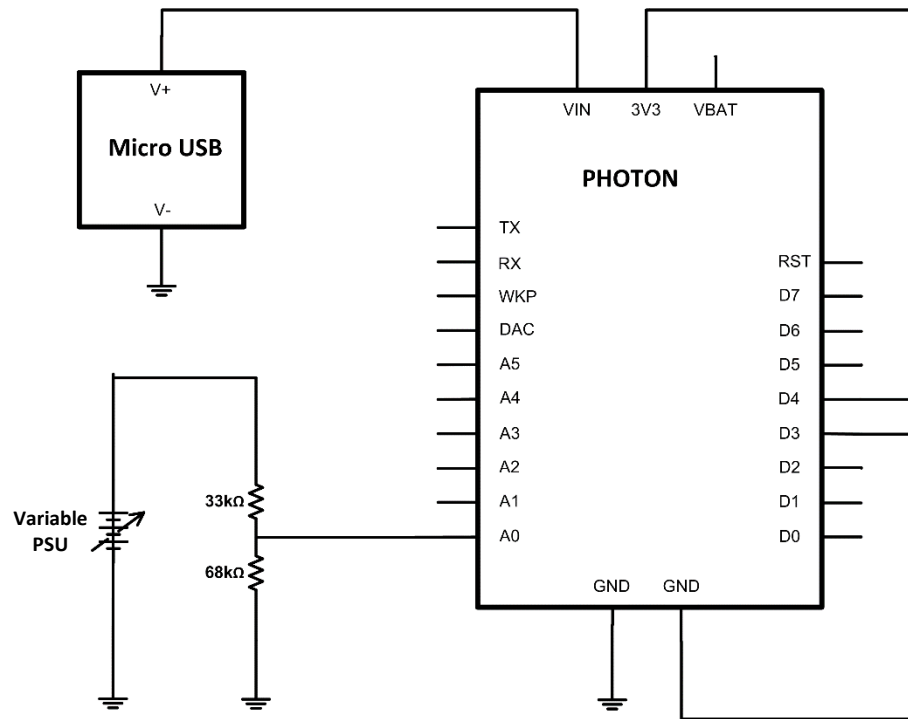


Figure 5.18: Components and Configuration used to Test the Battery Monitoring Feature

A variable PSU was applied as an input to the voltage monitoring circuit to simulate the behaviour of the battery. Additionally, 2 wires that were connected to digital input pins D3 and D4 were connected to 3.3 V or 0 V of the Photon, as required by the testing procedure, in order to simulate the high and low resistance states of the STAT pins, respectively. The segment of code shown in **Code Listing 5.7**, that was applicable, used the updated battery voltage range:

```
void BatteryStatus(){

    BatteryVoltage=((analogRead(BatteryPin)*4.2)/3510))+0.015;
    //Calibrated battery voltage measurement
    Serial.println("Battery voltage: %0.2f V", BatteryVoltage);

    //Battery voltage under normal and non-charge conditions
    if((digitalRead(STAT1)==HIGH && digitalRead(STAT2)==HIGH) ||
        (digitalRead(STAT1)==LOW && digitalRead(STAT2)==LOW)){
```

```

//Check battery voltage
if(BatteryVoltage<=3.36){           //Battery critically low condition

    Serial.println("Battery critically low. Please connect charger
immediately!");

}
else if(BatteryVoltage<=3.4 ){//Battery low condition at 3.4V

    Serial.println("Battery is low. Charge battery");

}
else if(BatteryVoltage>3.40){ //Battery normal condition

    Serial.println("Battery is Normal");

}
}

//Battery charging conditions
if(digitalRead(STAT1)==LOW && digitalRead(STAT2)==HIGH){

    Serial.printlnf("\nBattery is charging");

}

// Battery charged conditions
if(digitalRead(STAT1)==HIGH && digitalRead(STAT2)==LOW){

    Serial.println("Battery is Charged");

}
}

```

Code Listing 5.7: Testing the Battery Status Monitoring Feature

The following conditions were created in order to verify the changes in battery status:

1. The voltage state of D3 and D4 were kept identical (both “High” and then both “Low”) while the potentiometer’s voltage was varied from 4.2 V to 3.36 V to test the “Battery normal”, “Battery low” and “Battery critically low” statuses;
2. The state of the input pins D3 and D4 were then varied to be in alternate states, 1 “High” while the other was “Low” and vice versa, to test the “Battery charging” and “Battery charged” statuses. Additionally, the illumination of the charge indication LED was observed under “Battery charging” conditions.

Since the use of the serial monitor requires a connection between the Photon and a computer via micro USB, and since the Photon cannot be connected to the micro USB and the battery simultaneously, further analysis was conducted when testing the mobile application interface which is discussed later in section 6.2.

5.12.2 Battery Status Test Result

It was observed that the minor noise affecting the ADC’s channel 0 caused small fluctuations in the battery voltage measurements that were produced, which in turn caused the charge status indications to fluctuate when transitioning between statuses. To remedy this, the battery voltage measurement is prevented from increasing, by adding the code segment shown in **Code Listing 5.8**, under the “Battery normal”, “Battery low”, “Battery critically low” conditions.

```
//To prevent battery fluctuation, the battery voltage is forced to be the
lowest state it is measured to be
if(BatteryVoltage <= BatteryVoltageComparison){

    BatteryVoltageComparison=BatteryVoltage;
    //Serial.println("BatteryVoltageComparison=BatteryVoltage");

}
else if(BatteryVoltage > BatteryVoltageComparison){

    BatteryVoltage=BatteryVoltageComparison;
```

```

        //Serial.println("BatteryVoltage=BatteryVoltageComparison");
    }

```

Code Listing 5.8: Code Used to Prevent Battery Fluctuations Under Non Charge Conditions

The “Battery normal”, “Battery low” and “Battery critically low” statuses were observed between the expected voltage ranges 4.2 V to 3.4 V, 3.4 V to 3.36 V and below 3.36 V respectively. The battery charge statuses also changed as expected: when pin D3 was “High” and D4 was “Low” the battery charging status was indicated and when pin D3 was “Low” and D4 was “High” the battery charged status was indicated. Furthermore, the charge indication LED was illuminated when the charger was plugged into the weather station’s USB-C port.

5.13 Firmware Execution Period

5.13.1 Firmware Execution Period Testing

This test was conducted after all firmware adjustments were completed. The time taken for 1 cycle of the firmware to be executed (firmware execution period) was measured, using the segment of code shown in **Code Listing 5.9**:

```

void AllFunctions(){

    BeginTime=millis();           //The starting time before execution begins
    Serial.printlnf("\nBeginTime: %i\n", BeginTime);

    //Weather Station Functions
    GetTimeStamp();
    Windsensor();
    GetVEML7700SensorData();
    RainSensor();
    GetBME280SensorData();
    DisplayOnSerialMonitor();
    SaveToSDCard();
    BlynkComms();                 //Called last to allow variables to be
    updated
}

```

```

PowerSaveMode();           //Checks if stop mode should be activated or
                             not

EndTime=millis();           //The time after the execution was completed
Serial.println("EndTime: %i", EndTime);
FirmwareExecutionPeriod=(EndTime-BeginTime)/1000;
Serial.println("\nFirmware execution period: %0.2f seconds\n",
FirmwareExecutionPeriod);
}

```

Code Listing 5.9: Execution Period of the Weather Station's Firmware

The weather station's firmware execution period was tested under the following conditions:

1. All sensors were connected to the weather station, the SD card was inserted into the SD card reader and the data logger was activated;
2. All sensors were disconnected from the weather station and the SD card was removed;
3. The weather station app disconnected from the weather station, with all the sensors and the SD card disconnected;
4. All sensors disconnected first, then with all the sensors connected and the battery voltage simulated to be below 3.36 V under both conditions (continuous notification information sent to Blynk).

The results were observed on the serial monitor and used to determine the execution frequency of the "AllSubroutines" subroutine.

5.13.2 Firmware Execution Period Test Result

The initial firmware execution period was found to be 1.98 seconds but after all adjustments were made to the firmware, the final execution period was found to be 2.05 seconds on average and never exceeded 3 seconds. It was therefore deemed appropriate for the timer controlling the "AllSubroutine" subroutine to keep a 3-second period. This would ensure the entire firmware cycle is completed before the next cycle is initiated.

5.14 Period of Operation

5.14.1 Period of Operation Testing

The weather station's period of operation was determined after all adjustments to the firmware were completed. The battery was fully charged before the station's period of operation was tested in each of the following two situations:

1. All the weather station's sensors were connected to the station and the data logger was activated. The spare pin D7 of the Photon was configured as an output and connected to the Photon's rain and wind speed input pins, D5 and D2 respectively. The segment of code shown in **Code Listing 5.10** was included in the firmware to simulate periodic wind speed and rainfall measurements:

```
void loop(){

    Blynk.run();    //Maintain connection to Blynk
    timer1.run();   //Keep timer1 running

    //To simulate the wind speed and rainfall measurements periodically
    counter3=counter3+1;
    if (counter3==100){

        digitalWrite(SimPin, HIGH); //SimPin is output pin D7

        //Serial.println("SimPin went high");
        counter3=0;

    }
    digitalWrite(SimPin, LOW);

}
```

Code Listing 5.10: Code Segment Used to Simulate Periodic Wind Speed and Rainfall Measurements

This code was included so that the test results are reliable and repeatable;

2. All the weather station's sensors except for the wind and rain sensors were connected to the station and the data logger was activated. The weather station was operated while cycling through sleep mode.

The weather station was operated continuously and the timestamped battery voltage level was logged to the micro SD card every 3 seconds.

5.14.2 Period of Operation Test Result

The results of the battery capacity tests are summarised in the graph shown in **Figure 5.19**.

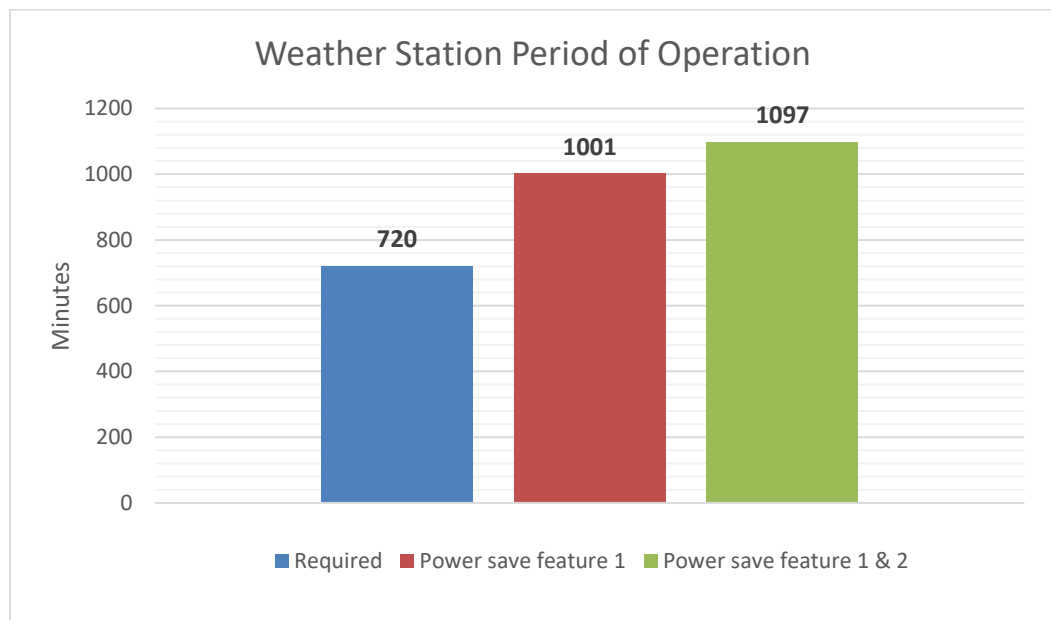


Figure 5.19: Graph Showing the Required and Achieved Periods of Operation

The 1050 mAh battery was able to power the station, with a voltage above 3.36 V, for 16 hours and 41 minutes versus 18 hours and 17 minutes while continuously cycling through sleep mode. Both periods of operation exceed the 12 hours (720 minutes) required by the design specification. Additionally, the weather station took 21 minutes to drop from 3.4 V to 3.36 V which was deemed to be an adequate amount of time for a charger to be sourced to initiate charging.

5.15 Problems Encountered and their Solutions

5.15.1 Wind Speed Sensor Testing

The lack of professional grade equipment necessitated the use of house-hold items to conduct wind speed testing. The original idea was to measure the wind speed with a wind tunnel setup that is shown in **Figure 5.20**.

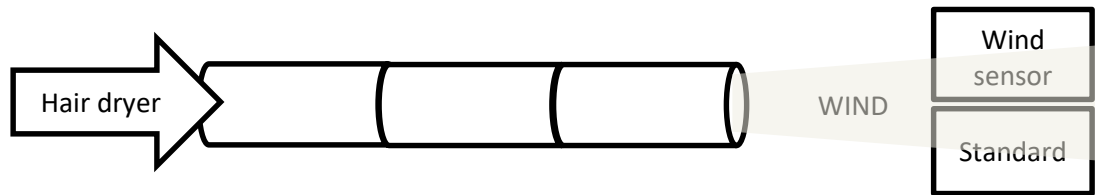


Figure 5.20: Initial Method for Wind Speed Sensor Testing

After several attempts, however, this method was found to be unreliable: since the 2 wind sensors were not of the same type, their focal points differed and since the wind produced could not envelope both sensors simultaneously, they experienced the same wind condition differently. The solution to this problem was to employ a wind source capable of producing a larger stream of wind that enveloped each sensor completely, such as a household fan as shown in **Figure 5.1**.

CHAPTER 6: MOBILE APPLICATION TESTING AND RESULTS

This chapter explains the tests that were conducted to verify the functionality of the weather station app and the results that were obtained. Changes that were implemented to improve the functionality of the app are also detailed in this chapter.

6.1. Weather Sensor Measurements and Connection Status

6.1.1 Testing

The weather station app was started and a connection to the weather station hardware was initiated, using the Bluetooth widget, under the following conditions:

1. With power save mode deactivated- all sensors connected to the weather station;
2. With power save mode activated- the rain and wind sensors disconnected from the station.

The sensors were systematically disconnected and reconnected to the weather station and their connection statuses as well as the measurements they produced were observed. This was to verify, not only if the information displayed was correct but, that the sensor data was displaying in the correct widget.

6.1.2 Results

Weather measurement and connection status information that was displayed on the app were found to be reliable in both test conditions. The mobile interface updated every 3 seconds. It was also observed that when the app connected to the weather station hardware, there was no clear indication to verify this, other than to observe minor changes and fluctuations in the sensor measurements. To resolve this, the `Blynk.notify("Weather Station: Online")` command was added to the "void setup()" subroutine which enabled a notification to be produced when the app first established a connection to the hardware.

6.2. Battery Status Monitoring

6.2.1 Testing

The battery voltage, battery charge indications, notifications and LED widget colours were tested while verifying the operating battery voltage range which is mentioned later. Thus, the changes in battery voltage and status were tested under the following conditions:

1. The “Battery normal”, “Battery low” and “Battery critically low” statuses were tested while the battery and charge controller combination was connected to the weather station;
2. When the battery was completely depleted then the charger was connected to the weather station, until the battery was fully charged, to test the “Battery charging” and “Battery charged” statuses;
3. The battery was then drained and when the battery voltage was depleted again the circuit configuration shown in **Figure 6.1** was applied:

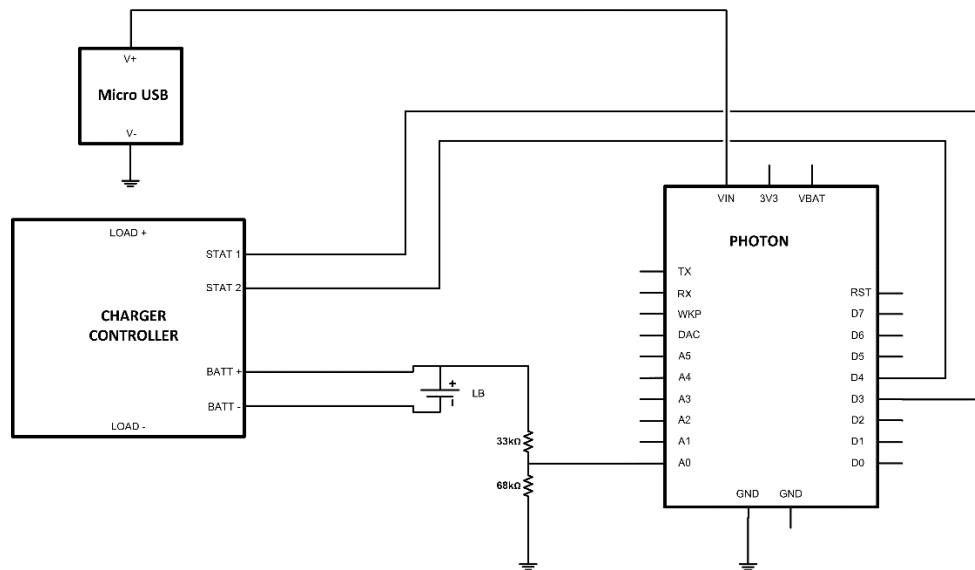


Figure 6.1: Circuit Used to Test the Charging Period of the Battery While it is Disconnected from the Load

The battery was prevented from powering the weather station and instead it was only connected to the battery voltage monitoring circuit. The charger was then connected to test the “Battery charging” and “Battery charged” statuses while the weather station was powered via the Photon’s micro USB port. The purpose of this test was to determine the battery charge period while the weather station was offline.

Conditions 1 and 2 were tested with and without the weather station cycling through sleep mode. The results of the battery voltage and status testing, under each test condition, were timestamped and logged to the SD card every 3 seconds using the segment of code in **Code Listing 6.1**:

```
void BatteryStatus(){

    BatteryVoltage=((analogRead(BatteryPin)*4.2)/3510))+0.015;
    //Calibrated battery voltage

    //To generate the timestamp
    t=Time.now();
    Seconds=Time.second(t);
    Minutes=Time.minute(t);
    Hours=Time.hour(t);
    Day=Time.day(t);
    Month=Time.month(t);

    //Battery normal, battery low or battery critically low condition
    if((digitalRead(STAT1)==HIGH && digitalRead(STAT2)==HIGH) ||
    (digitalRead(STAT1)==LOW && digitalRead(STAT2)==LOW)){

        Blynk.setProperty(V13, "label", "Battery voltage");

        BatteryChargedNotification=1;
        BatteryChargingNotification=1;

        //To preevent battery fluctuations
        if(BatteryVoltage <= BatteryVoltageComparison){

            BatteryVoltageComparison=BatteryVoltage;

            //Serial.println("BatteryVoltageComparison=BatteryVoltage");

        }
        else if(BatteryVoltage > BatteryVoltageComparison){

            BatteryVoltage=BatteryVoltageComparison;
```

```

        //Serial.println("BatteryVoltage=BatteryVoltageComparison");
    }

    //Check battery voltage
    if(BatteryVoltage<=3.36){    //Critically low voltage
    condition

        Blynk.setProperty(V12, "color", "#F00606");    //Red LED
        Blynk.setProperty(V12, "label", "Battery low");    //Set
        the LED
        widget to display that the battery is low
        //Serial.println("Battery critically low. Please connect
        Charger immediately!");
        Blynk.notify("Battery critically low. Please connect
        Charger immediately!"); //Notification displayed
        constantly
        printToCard.printf("\n%i/%i/%i  %i:%i:%i, Battery voltage is
        <3.36V", Day, Month, Year, Hours, Minutes, Seconds);

    }

    //Low voltage condition at 3.4V
    else if(BatteryVoltage<=3.4 && BatteryLowNotification==1){

        Blynk.setProperty(V12, "color", "#F00606");    //Red LED
        Blynk.setProperty(V12, "label", "Battery low");    //Set
        The widget to display that the battery is low
        //Serial.println("Battery is low. Charge battery");
        Blynk.notify("Battery is Low. Please connect charger!");
        //Notification displayed once
        BatteryLowNotification=0;
        printToCard.printf("\n%i/%i/%i  %i:%i:%i, Battery is
        low", Day, Month, Year, Hours, Minutes, Seconds);

    }

    else if(BatteryVoltage>3.4){    //Normal voltage condition

        Blynk.setProperty(V12, "color", "#23C48E");    //Green LED

```

```

        Blynk.setProperty(V12, "label", "Battery normal"); //Set
        the LED widget to display that the battery is normal
        //Serial.println("Battery normal");
        printToCard.printf("\n%i/%i/%i  %i:%i:%i, Battery is
        Normal", Day, Month, Year, Hours, Minutes, Seconds);

    }
}
//Battery charging condition
if(digitalRead(STAT1)==LOW && digitalRead(STAT2)==HIGH){

    BatteryChargedNotification=1;
    BatteryLowNotification=1;

    //Display battery charging notification once
    if(BatteryChargingNotification==1){

        Blynk.notify("Battery charging");
        BatteryChargingNotification=0;

    }
    //Update battery status widgets
    Blynk.setProperty(V12, "color", "#DFED00"); //Yellow LED
    Blynk.setProperty(V12, "label", "Battery charging"); //Set
    the LED widget to display that the battery is charging
    Blynk.setProperty(V13, "label", "Battery charge voltage");
    printToCard.printf("\n%i/%i/%i,  %i:%i:%i, Battery is
    charging", Day, Month, Year, Hours, Minutes, Seconds);
}
// Battery charged conditions
if((digitalRead(STAT1)==HIGH && digitalRead(STAT2)==LOW) ||
(digitalRead(STAT1)==LOW && digitalRead(STAT2)==HIGH)){

    //Update the battery status widgets
    Blynk.setProperty(V12, "color", "#1A6BF4"); //Blue LED
    Blynk.setProperty(V12, "label", "Battery charged"); //Set
    the LED widget to state that the battery is charged
    Blynk.setProperty(V13, "label", "Battery voltage");
    //Serial.println("Battery is Charged");
    printToCard.printf("\n%i/%i/%i,  %i:%i:%i, Battery has

```



```

        fully charged", Day, Month, Year, Hours, Minutes, Seconds);

    //Display battery charged notification once
    BatteryChargingNotification=1;

    //Executes once
    if(BatteryChargedNotification==1){

        Blynk.notify("Battery charged");

    }

}

//Updates the battery voltage on the display widget
if(BatteryVoltage<=3.36){

    Blynk.virtualWrite(V13, BatteryVoltageLow); //Display
    "<3.36V" in battery voltage display widget
    //Serial.println("Battery voltage is <3.36V");
    printToCard.println("Battery voltage is <3.36V");
}
else{

    Blynk.virtualWrite(V13, BatteryVoltage);
    //Serial.println("Battery voltage is %0.2f",
    BatteryVoltage);
    printToCard.println("Battery voltage: %0.2f V",
    BatteryVoltage);

}

}
}

```

Code Listing 6.1: To Test the Battery Status Monitoring Feature on the App and Log the Battery Voltage and Status Data

6.2.2. Results

The tests involving the battery and charge controller combination were conducted and the results are as follows:

1. Both with and without cycling through sleep mode: the “Battery normal”, “Battery low” and “Battery critically low” statuses occurred as expected – when the battery voltage was above 3.4 V, below 3.4 V, below 3.36 V respectively;
2. The “Battery charging” and “Battery charged” statuses occurred as expected while the weather station was cycling through sleep mode: immediately after the charger was connected and when the battery was fully charged, respectively. With sleep mode not active, the “Battery charging” status occurred again as expected but the “Battery charged” status was never reached even though the battery was fully charged. It was realised after investigating, that if the charge controller is powering the load and charging the battery with a constant current level that prevents the charge controller’s termination current (I_{TERM}) from dropping to below 5% of the charge current (I_{REG}), then the state of the STAT pins will never change even if the battery is fully charged. Since the charge controller’s STAT pins could not change state, when the weather station was not cycling through sleep mode, the “Battery charged” status was achieved through the implementation of a timer. The amount of time taken for the battery status to transition from “Battery charging” to “Battery charged”, while cycling through sleep mode, was available through the timestamped battery status entries logged onto the micro SD card. It was convenient to choose 4.2 V as the reference point since it was the maximum battery charge voltage measurable by the ADC and therefore this voltage level had the minimum charge time to completion. The battery voltage reached 4.2 V after 2 hours and 26 minutes and the complete battery charge time, according to the logged data, was calculated to be 3 hours and 41 minutes. Therefore, the Photon was programmed to wait 1 hour and 15 minutes, after the battery voltage reaches 4.2 V before the “Battery charged” status is displayed. The battery status tests were then repeated, using the segment of code shown in **Code Listing 6.2**:

```
void BatteryStatus(){  
  
    //Calibrated battery voltage  
    BatteryVoltage=((analogRead(BatteryPin)*4.2)/3510))+0.015;
```

```

//To generate the time stamp
t=Time.now();
Seconds=Time.second(t);
Minutes=Time.minute(t);
Hours=Time.hour(t);
Day=Time.day(t);
Month=Time.month(t);
Year=Time.year(t);

//Battery normal, battery low or battery critically low
condition
if((digitalRead(STAT1)==HIGH && digitalRead(STAT2)==HIGH) ||
(digitalRead(STAT1)==LOW && digitalRead(STAT2)==LOW)){

    Blynk.setProperty(V13, "label", "Battery voltage");

    BatteryChargedNotification=1;
    BatteryChargingNotification=1;

    //To prevent battery fluctuations
    if(BatteryVoltage <= BatteryVoltageComparison){

        BatteryVoltageComparison=BatteryVoltage;
        //Serial.println("BatteryVoltageComparison=BatteryVoltage");

    }
    else if(BatteryVoltage > BatteryVoltageComparison){

        BatteryVoltage=BatteryVoltageComparison;
        //Serial.println("BatteryVoltage=BatteryVoltageComparison");

    }

    //Check battery voltage
    if(BatteryVoltage<=3.36){ //Critically low voltage
condition

```

```

    Blynk.setProperty(V12, "color", "#F00606"); //Red LED
    Blynk.setProperty(V12, "label", "Battery low"); //Set
    the LED widget to display that the battery is low
    //Serial.println("Battery critically low. Please
    connect Charger immediately!");
    Blynk.notify("Battery critically low. Please connect
    charger immediately!"); //Notification displayed
    constantly
    printToCard.printf("\n%i/%i/%i %i:%i:%i, Battery
    voltage is <3.36V", Day, Month, Year, Hours, Minutes,
    Seconds);
}
//Low voltage condition at 3.4V
else if(BatteryVoltage<=3.4 && BatteryLowNotification==1){

    Blynk.setProperty(V12, "color", "#F00606"); //Red LED
    Blynk.setProperty(V12, "label", "Battery low"); //Set
    the widget to display that the battery is low
    //Serial.println("Battery is low. Charge battery");
    Blynk.notify("Battery is Low. Please connect
    charger!"); //Notification displayed once

    BatteryLowNotification=0;
    printToCard.printf("\n%i/%i/%i %i:%i:%i, Battery
    voltage is low", Day, Month, Year, Hours, Minutes,
    Seconds);
}
//Normal voltage condition
else if(BatteryVoltage>3.40){

    Blynk.setProperty(V12, "color", "#23C48E"); //Green
    LED
    Blynk.setProperty(V12, "label", "Battery normal");
    //Set the LED widget to display that the battery is
    normal

    //Serial.println("Battery normal");

```

```

        printToCard.printlnf("\n%i/%i/%i  %i:%i:%i, Battery
        voltage is normal", Day, Month, Year, Hours, Minutes,
        Seconds);

    }
}
//Battery charging condition
if(digitalRead(STAT1)==LOW && digitalRead(STAT2)==HIGH &&
ChargeCounter>0){

    BatteryVoltageComparison=5;    //To ensure that this
    variable will be set the new battery voltage under normal
    battery operating conditions
    BatteryChargedNotification=1;
    BatteryLowNotification=1;

    //Display battery charging notification once
    if(BatteryChargingNotification==1){

        Blynk.notify("Battery charging");
        BatteryChargingNotification=0;

    }

    //Update battery status widgets
    Blynk.setProperty(V12, "color", "#DFED00");    //Yellow LED
    Blynk.setProperty(V12, "label", "Battery charging");
    //Set the LED widget to display that the battery is charging
    Blynk.setProperty(V13, "label", "Battery charge voltage");
    printToCard.printlnf("\n%i/%i/%i,  %i:%i:%i, Battery is
    charging", Day, Month, Year, Hours, Minutes, Seconds);

    //Ensures that the changing of the charge status by timing
    is only initiated when the power saving feature is not
    active
    if(BatteryVoltage>=4.2 && (PowerSave1==0 ||
    PowerSave2==0)){

        t=Time.now();    //Setting 't' to be the current time

        //Start the charging timer

```

```

        if(ChargeCounter==4501){

            ChargeStartTime=t;
            ChargeCounter=4500;    //4500 seconds is equivalent
                                   to 1 hour and 15 minutes

            //Determine the end of the charging period
            ChargeEndTime=ChargeStartTime+ChargeCounter;

        }
        else{

            //When the charge time is completed
            if(ChargeEndTime<=t){

                ChargeCounter=-1;

            }

        }

    }

}

// Battery charged conditions
if((digitalRead(STAT1)==HIGH && digitalRead(STAT2)==LOW) ||
    (ChargeCounter<=0 && digitalRead(STAT1)==LOW &&
    digitalRead(STAT2)==HIGH)){

    //Resets the 'ChargeCounter' if fully charged state was
    achieved by STAT pins
    if(digitalRead(STAT1)==HIGH && digitalRead(STAT2)==LOW){

        ChargeCounter=4501;

    }

    //Update the battery status widgets
    Blynk.setProperty(V12, "color", "#1A6BF4");    //Blue LED on
    Blynk.setProperty(V12, "label", "Battery charged");    //Set
    the LED widget to state that the battery is charged

```

```

Blynk.setProperty(V13, "label", "Battery voltage");
//Serial.println("Battery is Charged");
printToCard.printf("\n%i/%i/%i, %i:%i:%i, Battery has
fully charged", Day, Month, Year, Hours, Minutes, Seconds);

//Display battery charged notification once
BatteryChargingNotification=1;

//Executes once
if(BatteryChargedNotification==1){

    Blynk.notify("Battery charged");

}

}
//Updates the battery voltage on the display widget
if(BatteryVoltage<=3.36){

    Blynk.virtualWrite(V13, BatteryVoltageLow); //Display
"<3.36V" in battery voltage display widget
    printToCard.printf("Battery voltage is <3.36V");
    //Serial.println("Battery voltage is <3.36V");

}
else{

    Blynk.virtualWrite(V13, BatteryVoltage);
    printToCard.printf("Battery voltage is %0.2f",
    BatteryVoltage);
    //Serial.printf("Battery voltage is %0.2f",
    BatteryVoltage);

}

}

```

Code Listing 6.2: Improved Segment of Test Code Used to Log Battery Voltage and Status Data

When the “Battery charged” state was indicated on the app then the weather station was switched “Off” immediately, dropping I_{TERM} below 5% of I_{REG} , and the charge controller’s on board 2 STAT LEDs were observed:

- The “CHRG” LED (which indicates that the battery is charging when it is illuminated) immediately switched “Off”;
- The “DONE” LED (which indicates that the battery has charged to completion when it is illuminated) instantly switched “On”.

This proved that this method is effective and reliable;

3. The amount of time taken to charge the battery fully in this configuration was calculated, using the timestamped battery status entries logged to the SD card, to be 3 hours 14 minutes.

6.3. Data Logger

6.3.1 Testing

The data logging feature was tested under the following conditions:

1. The push button widget was toggled to enable data logging while the micro SD card was present in the SD card reader;
2. The push button widget was toggled to disable data logging while the micro SD card was present in the SD card reader;
3. The micro SD card removed from the SD card reader while data logging was active;
4. The push button widget was toggled to enable data logging while no micro SD card was present in the SD card reader;
5. The push button widget was toggled to enable data logging while a faulty SD card was inserted into the SD card reader.

The 5 test conditions described above were conducted both with and without the weather station cycling through sleep mode using the code segment shown in **Code Listing 6.3**.

```
//Subroutine that enables/disables data logging to the SD card
void SaveToSDCard(){

    //Execute when the data logger feature is activated and the battery
    voltage is not critically low
```



```

if(DataLogger==1 && BatteryVoltage>3.36){

    //Checks if the SD card is connected
    if (digitalRead(DetectPin2)==HIGH){

        //Seperate the data batches stored on the SD card
        if(CheckSDCard==1){

            printToCard.println("\nNew batch of weather data,Date,
            Time, Temperature(C),Humidity(%),Pressure(hPa),Wind
            speed(km/hr),Wind direction, Rainfall(mm/hr)"); //This
            will run once only when the SD is first activated
            //Serial.println("\nNew batch of weather data");

            if(!printToCard.getLastBeginResult()){

                Blynk.notify("Data logger initializing, please
                wait...");
                //Serial.println("Data logger initializing, please
                wait...");
                CheckSDCard=1;
                SDCount++;

                if(SDCount==10){ //If the data logger fails after
                10 attempts then a fault indication is produced

                    Blynk.notify("Data logger failed");
                    Blynk.virtualWrite(V14,0);
                    DataLogger=0;;
                    SDCount=0;

                }
            }
        }
        else{

            Blynk.notify("Data logger: Enabled");
            //Serial.println("Data logger: Enabled");
            CheckSDCard=0;
            SDCount=0;
        }
    }
}

```

```

    }
}
else if(CheckSDCard==0){    //Log timestamped data to SD
card

    t=Time.now();
    Seconds=Time.second(t);
    Minutes=Time.minute(t);
    Hours=Time.hour(t);
    Day=Time.day(t);
    Month=Time.month(t);
    Year=Time.year(t);

    printToCard.println(" ,%i/%i/%i,%i:%i:%i,%0.2f,%0.2f,
%0.2f,%0.2f,%s,%0.2f", Day, Month, Year, Hours,
Minutes, Seconds, Temperature, Humidity, Pressure, Speed,
Direction, RainFall);
    //Serial.println("%i/%i/%i  %i:%i:%i  Temperature: %0.2f
C, Humidity: %0.2f %, Pressure: %0.2f hPa, Windspeed:
%0.2f km/hr, Wind direction: %s, Rainfall: %0.2f mm", Day,
Month, Year, Hours, Minutes, Seconds, Temperature,
Humidity, Pressure, Speed, Direction, RainFall);

}
}

else{    //Executes if the data logging is active/activated but no
SD card is present in the SD card reader

    //Serial.println("No SD card present");
    Blynk.notify("No SD card present");
    Blynk.virtualWrite(V14,0);
    DataLogger=0;
    SDCount=0;

}
}

//Disable data logger from activating when battery is critically low

```

```

else if(DataLogger==1 && BatteryVoltage<=3.36){

    printToCard.println("Data logger de-activated. Battery is
critically low. ");
    Blynk.notify("Data logger de-activated. Battery is critically low.
");
    Blynk.virtualWrite(V14,0);
    DataLogger=0;

}

}
//Executes when the data logger PB is toggled on the app
BLYNK_WRITE(V14){

    DataLogger = param.asInt();    //The state of the PB

    //Activate data logging
    if(DataLogger==HIGH){

        CheckSDCard=1;

    }
    //De-activate data logging
    else{

        Blynk.notify("Data logger: Disabled");
        //Serial.printlnf("Save data to SD card: Disabled");

    }

}
}

```

Code Listing 6.3: Controlling the Data Logger from the Mobile App

6.3.2. Results

Data logging and generation of the respective notifications occurred as expected, under each of the conditions, while the weather station was not cycling through sleep mode. However,

the push button presses were sometimes missed while the weather station was cycling through sleep mode. The code shown in **Code Listing 6.4** was implemented to solve this:

```
//Checks if sleep mode should be implemented
void PowerSaveMode(){

    if(PowerSave1==1 & PowerSave2==1){    //Power saving mode activated
        when both the rain sensor and the windsensor are disconnected from the
        station

        System.sleep(DetectPin1,RISING,SleepTime);
        CheckDataLoggerPB();    //Update the state of the data logger PB
        after returning from sleep/stop mode

    }

}
```

Code Listing 6.4: Responsible for Maintaining the State of the Push Button Widget

When sleep mode is enabled and the push button widget is toggled while the weather station is asleep, the push button press is missed and the weather station updates the state of the push button widget to the last state it received. This prevents false indications.

6.4 Communication Range Between the Weather Station Hardware and the App

6.4.1. Testing

Testing the communication range between the weather station hardware and mobile application involved separating the 2 devices by 1 metre intervals until a communication link between them was not able to be established. The communication range was tested under 2 conditions:

1. The path between the devices was kept clear of any obstacles;
2. The path between the devices was obstructed by obstacles such as trees, cars, walls and people.

The testing of the communication range was conducted last, after all the hardware components were installed into the enclosure and the project was completed. This was in order to take into consideration the barrier imposed by the enclosure on the transmission range of the JDY08 BLE module.

6.4.2. Results

It was found that the weather station was able to communicate reliably with the mobile app up to:

- 24 metres on an unobstructed path;
- 18 metres on the path that required the signal to travel around obstacles.

6.5. Problems Encountered and their Solutions

6.5.1 The Operating Battery Voltage Range

The battery status testing procedure mentioned here, which required the use of the circuit configuration shown in **Figure 6.2**, was applied to test the occurrence of notifications as well as the display and LED widget indications under the appropriate 3.6 V to 4.2 V operating battery voltage and charge conditions.

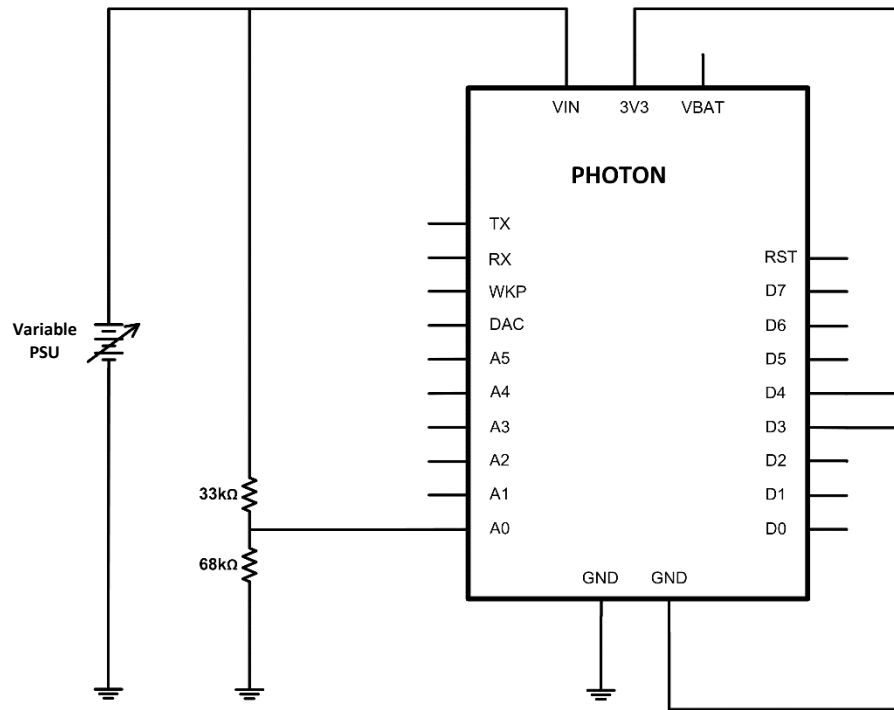


Figure 6.2: Testing the Minimum Operating Voltage of the Weather Station

This test circuit was used in conjunction with the segment of code shown in **Code Listing 6.5** to verify the weather station's operating voltage range.

```
void BatteryStatus(){

    BatteryVoltage=(((analogRead(BatteryPin)*4.2)/3510))+0.015;
    //Calibrated battery voltage

    //Battery normal, battery low or battery critically low condition
    if((digitalRead(STAT1)==HIGH && digitalRead(STAT2)==HIGH) ||
    (digitalRead(STAT1)==LOW && digitalRead(STAT2)==LOW)){

        Blynk.setProperty(V13, "label", "Battery voltage");

        BatteryChargedNotification=1;
        BatteryChargingNotification=1;

        //To prevent battery fluctuations
        if(BatteryVoltage <= BatteryVoltageComparison){

            BatteryVoltageComparison=BatteryVoltage;
        }
    }
}
```

```

        //Serial.println("BatteryVoltageComparison=BatteryVoltage");

    }
    else if(BatteryVoltage > BatteryVoltageComparison){

        BatteryVoltage=BatteryVoltageComparison;
        //Serial.println("BatteryVoltage=BatteryVoltageComparison");

    }

    //Check battery voltage
    if(BatteryVoltage<=3.65){    //Critically low voltage condition

        Blynk.setProperty(V12, "color", "#F00606");
        //Red LED
        Blynk.setProperty(V12, "label", "Battery low"); //Set the
        LED widget to display that the battery is low
        //Serial.println("Battery critically low. Please connect
        charger immediately!");
        Blynk.notify("Battery critically low. Please connect charger
        immediately!"); //Notification displayed constantly

    }
    else if(BatteryVoltage<=3.7 && BatteryLowNotification==1){ //Low
    voltage condition at 3.4V

        Blynk.setProperty(V12, "color", "#F00606"); //Red LED
        Blynk.setProperty(V12, "label", "Battery low");
        //Set the widget to display that the battery is low
        //Serial.println("Battery is low. Charge battery");
        Blynk.notify("Battery is Low. Please connect charger!");
        //Notification displayed once
        BatteryLowNotification=0;

    }
    else if(BatteryVoltage>3.7){    //Normal voltage condition

        Blynk.setProperty(V12, "color", "#23C48E"); //Green LED
        Blynk.setProperty(V12, "label", "Battery normal"); //Set
        the LED widget to display that the battery is normal
    }

```

```

        //Serial.println("Battery normal");

    }
}

//Battery charging condition
if(digitalRead(STAT1)==LOW && digitalRead(STAT2)==HIGH){

    BatteryChargedNotification=1;
    BatteryLowNotification=1;

    //Display battery charging notification once
    if(BatteryChargingNotification==1){

        Blynk.notify("Battery charging");
        BatteryChargingNotification=0;

    }
    //Update battery status widgets
    Blynk.setProperty(V12, "color", "#DFED00"); //Yellow LED
    Blynk.setProperty(V12, "label", "Battery charging");
    //Set the LED widget to display that the battery is charging
    Blynk.setProperty(V13, "label", "Battery charge voltage");

}

// Battery charged conditions
if((digitalRead(STAT1)==HIGH && digitalRead(STAT2)==LOW) ||
(digitalRead(STAT1)==LOW && digitalRead(STAT2)==HIGH)){

    //Update the battery status widgets
    Blynk.setProperty(V12, "color", "#1A6BF4"); //Blue LED on
    Blynk.setProperty(V12, "label", "Battery charged"); //Set the
    LED widget to state that the battery is charged
    Blynk.setProperty(V13, "label", "Battery voltage");
    //Serial.println("Battery is Charged");

    //Display battery charged notification once
    BatteryChargingNotification=1;

    //Executes once
    if(BatteryChargedNotification==1){

```



```

        Blynk.notify("Battery charged");

    }

}

//Updates the battery voltage on the display widget
Blynk.virtualWrite(V13, BatteryVoltage);
//Serial.println("Battery voltage is %0.2f", BatteryVoltage);

}

```

Code Listing 6.5: Test the Minimum Operating Voltage of the Weather Station

The voltages of the variable PSU were decreased from 4.2 V to 3.7 V, then to 3.65 V and thereafter in steps of 0.05 V until a voltage level was reached whereby the weather station's operation ceased. Voltages produced by the PSU were measured by a Fluke 177 multimeter and the multimeter's readings were compared to those made by the Photon. All notifications and indications were produced at the expected voltage levels and under the expected charge conditions. However, it was discovered that the Photon continued to function at a voltage level as low as 3 V, equivalent to the Li-ion battery's cut-off voltage, even though the lowest expected operating voltage was 3.6 V. This meant that the Photon was operating with an input voltage lower than 3.3 V which would adversely impact the accuracy of its ADC since the input voltage is used as a reference when producing measurements [51]. It was observed that when the battery voltage dropped below 3.35 V the measurement produced by the ADC became unreliable. This posed a problem for the battery voltage monitoring and the wind direction sensor measurements because their operation relies on the accuracy of the Photon's ADC. To resolve this, when the battery voltage drops below 3.36 V, the Photon is programmed to block:

- The wind direction measurements;
- The battery voltage measurements;
- The data logging feature from activating so that the incorrect wind direction measurements are not logged.

The weather station's minimum operating battery voltage level was also changed from 3.6 V to 3.36 V to take advantage of the extra minutes of reliable weather station operation that can be achieved when the battery voltage is drops below 3.6 V.

6.5.2. Weather Station Hardware and App Connection Interruption

While conducting the testing it was observed that the connection between the app and hardware is broken when the app is closed or run in the background of the mobile device. When the app was re-opened, it automatically reconnected to the weather station hardware but:

- It produced no clear indication to confirm this;
- The state of the push button widget responsible for data logging did not retain its last toggled state;
- The battery status notifications were missed if they occurred while the weather station app was disconnected from the hardware.

These were resolved by incorporating the “BLYNK_CONNECTED()” subroutine into the firmware, as shown in **Code Listing 6.6**, which is executed each time a connection between the hardware and app is established:

```
//Executes when the weather station hardware connects to the app
BLYNK_CONNECTED(){

    Blynk.notify("Weather Station: Connected");
    //Serial.println("Weather Station: Connected");

    CheckDataLoggerPB(); //Update the state of the Data logger PB on
    the mobile app

    //Ensures that the battery status notifications occur upon
    reconnection to the app so that they are not missed
    BatteryChargedNotification=1;
    BatteryChargingNotification=1;
    BatteryLowNotification=1;

}

//Checks the last known state of the data logger PB widget and updates it
on the app
void CheckDataLoggerPB(){

    if(DataLogger==0){
```

```
        Blynk.virtualWrite(V14,0);  
  
    }  
    else if(DataLogger==1){  
  
        Blynk.virtualWrite(V14,1);  
  
    }  
  
}
```

Code Listing 6.6: Responsible for Producing App Notifications and Updating the State of the Data Logger Push Button Widget After Weather Station App-Hardware Reconnection

CHAPTER 7: CONCLUSION AND RECOMMENDATIONS

7.1. Conclusion

The portable weather monitoring system that is presented in this report was observed to be operational for a minimum period of 16 hours and 41 minutes which exceeds the design requirement's 12-hour period of operation. The weather sensors' measurements were verified over their operating ranges:

- The BME280's temperature, pressure and humidity measurements were tested to produce maximum deviations of 1.4 °C, 1.46 hPa and 1.75% which is within the required tolerances of 2 °C, 10 hPa and 5% respectively;
- The Davis wind direction sensor was tested and calibrated to achieve wind direction measurements that were within 4° to that of the standard which is less than the 5° tolerance required by the design specifications;
- The 8.3% maximum deviation produced by the Davis wind speed sensor did not exceed the 10% tolerance required by the design brief;
- The rain sensor measurements were tested to achieve a 0.8 mm/hr deviation compared to the required 2mm/hr tolerance.

The weather station app was designed to be user friendly and was tested to be functioning as intended. Therefore, from the results that were obtained, it was concluded that this project fulfils the criteria required by the design specifications and would be found useful to various industries that require weather monitoring systems in South Africa.

7.2. Recommendations

Since this system was designed to operate within the extreme weather conditions in South Africa only, it is recommended that due precaution be taken when using the weather station elsewhere to prevent it from being exposed to damaging weather conditions that exceed its measuring ability. Additionally, the following recommendations have been identified to improve the design of the weather station:

- A second reed switch can be incorporated into the wind speed sensor circuit which will double the sensor's resolution to 0.453 km/hr;

- An app that does not require an internet connection to function over Bluetooth should be used, such as an app developed using the Virtuino platform;
- The newly released Particle Argon could be used as the Photon of choice, instead of the Particle Photon, since it consists of both an on-board BLE module and charging circuitry [52]. Furthermore, this Photon has mesh networking capabilities which means two or more weather stations can be installed and linked together in an area and their weather measurements can be averaged to achieve better weather measurement resolution;
- Replace Adafruit USB Lilon/LiPoly charge controller, which is currently used by this weather station, with an Adafruit solar charge controller [53]. This will give the weather station the additional ability to be recharged and powered by a suitable solar panel.

REFERENCES

- [1] A comparison between Bluetooth and Wi-Fi communication mediums available at:
<https://itstillworks.com/bluetooth-vs-wifi-power-consumption-17630.html> [21 July 2018]
- [2] Weather trends in South Africa available at:
<http://www.weathersa.co.za/learning/climate-questions/39-what-are-the-temperature-rainfall-and-wind-extremes-in-sa> [3 August 2018]
- [3] Lowest temperature ever recorded in South Africa available at:
https://en.m.wikipedia.org/wiki/Sutherland,_Northern_Cape [3 August 2018]
- [4] Formula that relates pressure and altitude available at:
https://www.engineeringtoolbox.com/air-altitude-pressure-d_462.html [3 August 2018]
- [5] Highest land elevation in South Africa available at:
<https://en.wikipedia.org/wiki/Mafadi> [3 August 2018]
- [6] Effect of temperature and moisture on the atmospheric pressure:
<http://peter-mulroy.squarespace.com/air-pressure> [4 August 2018]
- [7] Competing weather station tolerances available at:
<https://www.netatmo.com/en-eu/weather/weatherstation/specifications> [4 August 2018]
- [8] Particle photon datasheet available at:
<https://docs.particle.io/datasheets/wi-fi/photon-datasheet/> [4 August 2018]
- [9] Particle Photon pinout diagram available at:
<https://www.electronicwings.com/particle/particle-photon-board> [28 August 2019]
- [10] BME280 Datasheet available at:
https://ae-bst.resource.bosch.com/media/_tech/media/datasheets/BST-BME280-DS002.pdf
[11 August 2018]

- [11] Davis Anemometer 6410 Specification sheet available at:
https://www.davisinstruments.com/product_documents/weather/spec_sheets/6410_SS.pdf
 [21 August 2018]

- [12] Ohms Law available at:
<https://www.physics.uoguelph.ca/tutorials/ohm/Q.ohm.intro.html> [21 August 2018]

- [13] Wind direction sensor potentiometer voltage available at:
<http://cactus.io/hooks/weather/anemometer/davis/hookup-arduino-to-davis-anemometer> [22 August 2018]

- [14] Maintenance required for the Davis 6410 wind sensor available at:
https://www.davisinstruments.com/product_documents/weather/Doc_Sensor-Maintenance.pdf [22 August 2019]

- [15] Product manual provided for the BGT WS-601ABS2 rain sensor by the manufacturer Beijing Guoxinhuayuan Technology Co.,Ltd

- [16] Bluetooth BLE vs Classic comparison available at:
<https://www.onsetcomp.com/content/bluetooth-low-energy-closer-look> [4 September 2018]

- [17] JDY-08 datasheet available at:
<https://fccid.io/2AM2YJDY-08/User-Manual/User-Manual-3511895.pdf> [8 September 2018]

- [18] Description of the Adafruit micro SD card module which is available at:
<https://www.adafruit.com/product/254> [27 February 2019]

- [19] Lithium ion battery characteristics available at:
https://batteryuniversity.com/learn/article/charging_at_high_and_low_temperatures [16 September 2018]

- [20] Relationship between the current consumption and battery capacity of a device available at:
https://www.batteryweb.com/pdf/inverter_battery_sizing_faq.pdf [27 January 2019]

- [21] 3.7V li-ion battery datasheet available at:

- <http://www.mantech.co.za/datasheets/products/063450-EIE-R0%5E1.pdf> [28 March 2019]
- [22] The Adafruit charge controller's MCP73833 chip datasheet available at:
<https://cdn-shop.adafruit.com/datasheets/MCP73833.pdf> [24 September 2018]
- [23] Voltage divider resistance that is suggested by the developers available at:
<https://community.particle.io/t/photon-a-d-pin-input-resistance/24129/5> [22 February 2019]
- [24] The reason for choosing the RJ45 connector available at:
https://library.e.abb.com/public/f4ed839d4e2039cfc1257bbf002af3bf/Sensor%20Accessories_1VLC000710%20Rev.-%20en%202011.07.pdf [23 April 2019]
- [25] The USB-C standard available at:
<https://mybroadband.co.za/news/hardware/266277-why-the-usb-type-c-port-is-so-awesome.html> [23 April 2019]
- [26] RJ45 wiring standard available at:
<https://acuitysupport.zendesk.com/hc/en-us/articles/210113548-What-is-the-difference-between-T568A-T568B-> [23 April 2019]
- [27] Blynk documentation available at:
<https://docs.blynk.cc/> [29 December 2018]
- [28] Data sheet for the 3.3V regulator available at:
<https://www.mantech.co.za/datasheets/products/LP2950.pdf> [18 April 2019]
- [29] Reed switch data sheet available at:
<https://www.mantech.co.za/datasheets/products/MKA-10110.pdf> [9 November 2018]
- [30] Wind sensor circuit diagram available at:
<http://www.lexingtonwx.com/anemometer/> [9 November 2018]
- [31] Information about micro SD card current spikes available at:
<https://community.nxp.com/thread/422802> [10 March 2019]

- [32] Trace width according to IPC-2152 standard available at:
<https://circuitmaker.com/blog/deciding-trace-width-part-1> [23 April 2019]
- [33] PCB design guidelines on the layout and orientation of components:
<https://www.autodesk.com/products/eagle/blog/pcb-layout-basics-component-placement/>
[23 April 2019]
- [34] PCB design guidelines on routing of traces available at:
<https://www.autodesk.com/products/eagle/blog/top-10-pcb-routing-tips-beginners/>
[23 April 2019]
- [35] Blynk Tabs widget available at:
<https://docs.blynk.cc/#widgets-interface-tabs> [20 July 2019]
- [36] Blynk Superchart widget available at:
<https://docs.blynk.cc/#widgets-displays-superchart> [15 January 2019]
- [37] Main PCB's enclosure dimensions available at:
<http://www.mantech.co.za/datasheets/products/AP-4%20ENCL.pdf>
[15 June 2019]
- [38] Conversion from miles to kilometres available at:
<https://www.rapidtables.com/convert/length/mile-to-km.html> [10 November 2018]
- [39] Relationship between period and frequency available at:
<https://study.com/academy/lesson/wave-period-definition-formula-quiz.html> [16 November 2018]
- [40] Typical switch debounce times available at:
<http://www.ganssle.com/debouncing.html> [16 November 2018]
- [41] Particle's mapping function available at:
<https://docs.particle.io/reference/device-os/firmware/photon/#map-> [26 November 2018]

- [42] Deactivating the Photon's P0 Wi-Fi module available at:
<https://docs.particle.io/reference/device-os/firmware/photon/#off-> [12 December 2018]
- [43] Particle documentation on sleep mode ("Stop Mode") available at:
<https://docs.particle.io/reference/device-os/firmware/photon/#sleep-sleep->
[13 March 2019]
- [44] The Blynk authorisation token available at:
<https://docs.blynk.cc/#getting-started-getting-started-with-the-blynk-app-4-auth-token>
[1 January 2019]
- [45] Importing text (.txt) files to Microsoft Excel available at:
<https://support.office.com/en-us/article/import-or-export-text-txt-or-csv-files-5250ac4c-663c-47ce-937b-339e391393ba?ui=en-US&rs=en-US&ad=US> [6 March 2019]
- [46] Example program to obtain Unix time via Blynk available at:
https://github.com/blynkkk/blynk-library/blob/master/examples/Widgets/RTC_Advanced/RTC_Advanced.ino [14 March 2019]
- [47] Particle functions to convert Unix time to standard date and time available at:
<https://docs.particle.io/reference/device-os/firmware/photon/#second-> [14 March 2019]
- [48] Equation that describes the relationship between temperature, atmospheric pressure and altitude above sea level available at:
https://en.wikipedia.org/wiki/Density_of_air#Altitude [18 March 2019]
- [49] The altitude above sea level of all locations in South Africa available at:
<http://en-za.topographic-map.com/places/Johannesburg-7811367/> [18 March 2019]
- [50] The amount of drift experienced by the Photon's RTC available at:
<https://community.particle.io/t/newb-questions-about-yet-another-photon-clock-but/49861>
[5 April 2019]
- [51] Application note that explains the ADC's dependency on the microcontroller's supply voltage available at:

https://www.st.com/content/ccc/resource/technical/document/application_note/group0/3f/4c/a4/82/bd/63/4e/92/CD00211314/files/CD00211314.pdf/jcr:content/translations/en.CD00211314.pdf [18 April 2019]

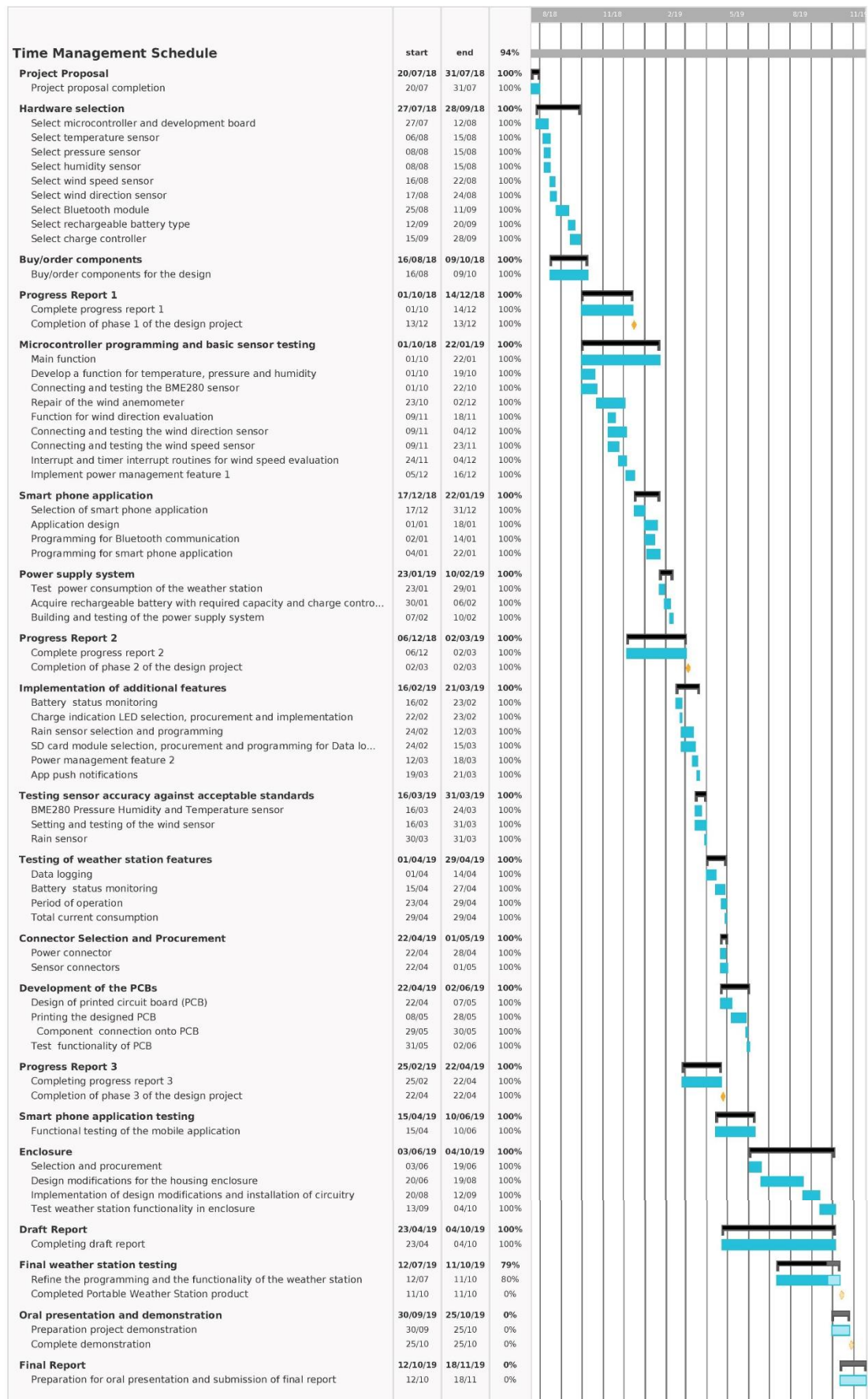
[52] The Particle Argon available at:

<https://docs.particle.io/argon/> [1 September 2019]

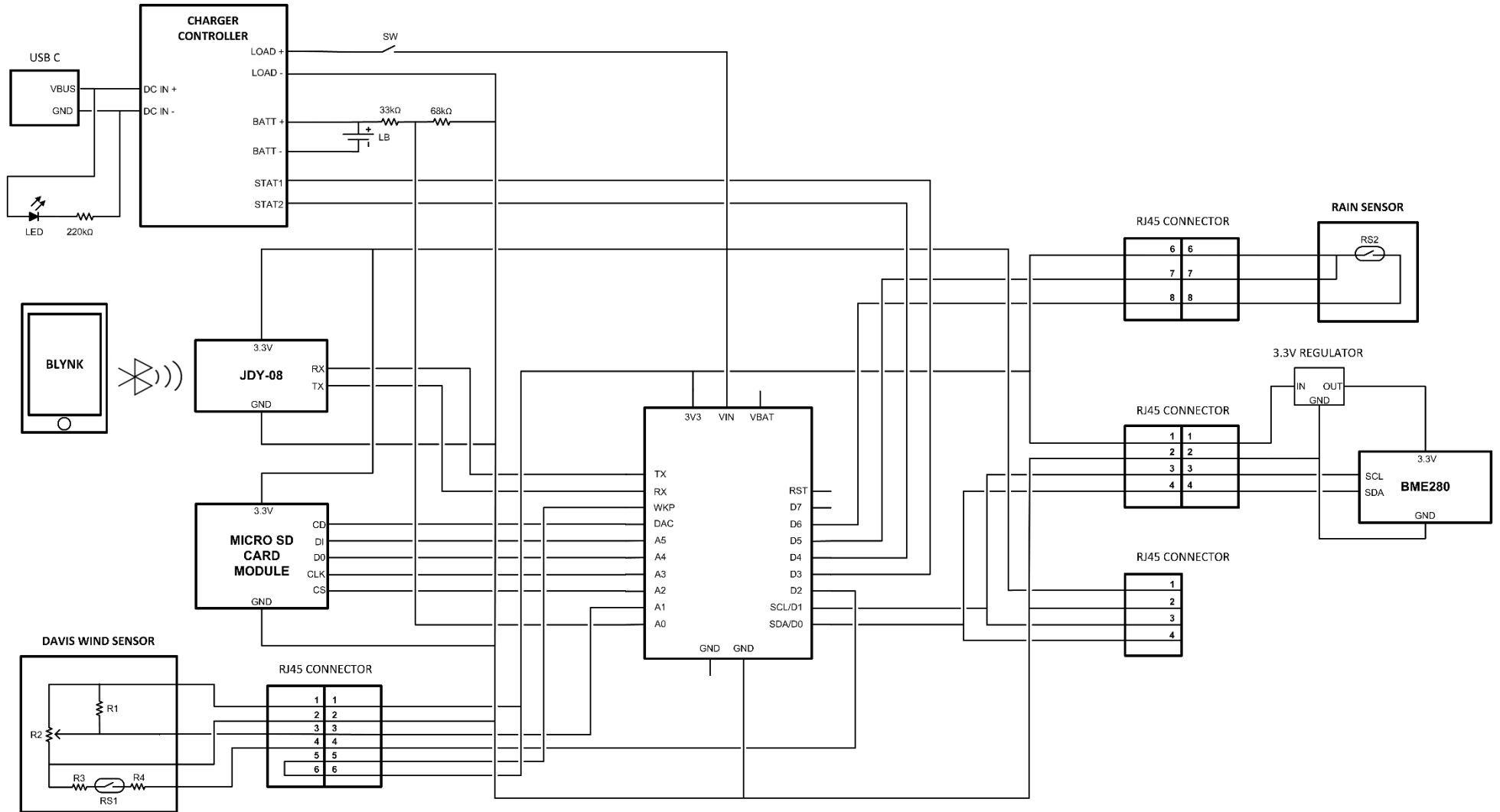
[53] The Adafruit USB/DC/Solar Lithium ion/Polymer charger-V2 available at:

<https://learn.adafruit.com/usb-dc-and-solar-lipoly-charger/using-the-charger?view=all#downloads> [3 September 2019]

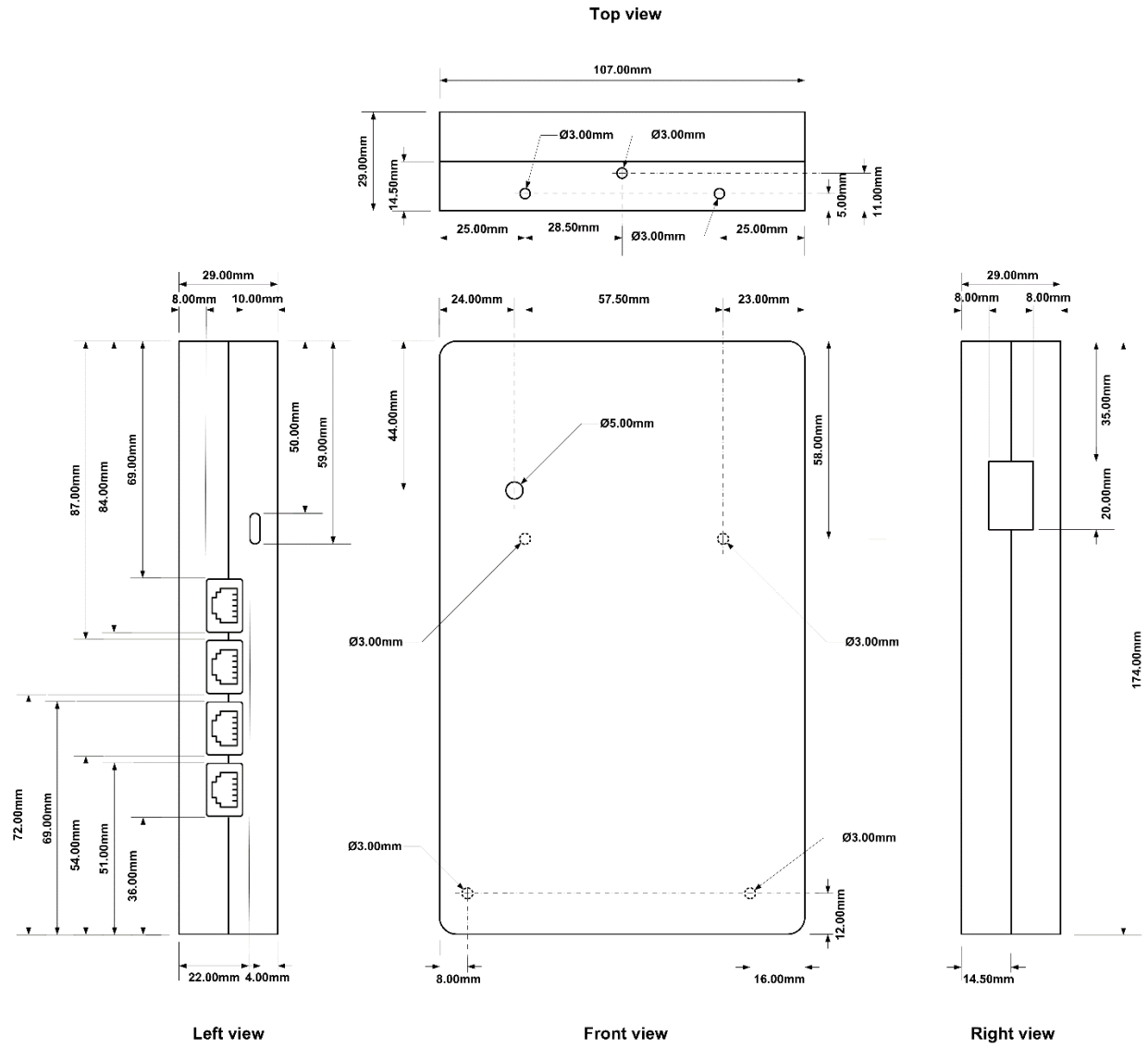
Annexure A: Time Management Schedule



Annexure B: Complete Circuit Schematic



Annexure C: Enclosure Modifications



Annexure D: Full Code Listing

```
/******
```

```
Weather Station Firmware Version 1.0.0
```

```
Written by: Muhammad Alli
```

```
Date created: 05-12-2018
```

```
Date modified last: 24-10-2019
```

```
The purpose of this program is to obtain, log and display weather sensor data  
to a mobile interface. This was achieved through the use  
of a Particle Photon microcontroller, 3 sensors, a micro SD card reader, a  
micro SD card and a Bluetooth Low Energy module:
```

- BME280 sensor (Temperature, pressure, humidity)
- Davis 6410 sensor (Wind speed and direction)
- BGT WS-601ABS2 sensor (Rain)
- Robotdyn SD card module (micro SD card reader)
- Any 32GB, or smaller, micro SD card
- JDY08 BLE module

```
The microcontroller interrogates the sensors and transmits the sensor data, via  
BLE, to a mobile application created on Blynk. The toggling of a push button on  
the mobile application controls the logging of timestamped weather data onto  
the micro SD card.
```

```

*****/

#include <BlynkSimpleSerialBLE.h>           //To enable Bluetooth communication
#include <CE_BME280.h>
#include<Particle.h>
#include<Wire.h>
#include "SdCardLogHandlerRK.h"
#define Addr 0x77                          //BME280 address (119 decimal)

//For Blynk communication
char auth[] = "c33adb3e3e8344b1bed36ec3c7c2317b"; //Unique authorisation token to allow communication between hardware
and mobile app
void BlynkComms(void);                     //Communicates hardware data to the mobile app
BlynkTimer timer1;                        //Create timer called timer1 to control push data to Blynk

//The controlling subroutine
void AllSubroutines(void);

//Subroutines, timers and variables for wind sensor operation
void WindsensorSetup(void);               //Subroutine to initialise the wind sensor
void Revolution(void);                    //Interrupt subroutine ISR
void WindDirection(void);                 //To get wind direction data

```



```

void Windsensor(void);
int WindspeedPin=D2;
int DebounceTime=0;
float SampleTime1=4000;
float Speed=0;
volatile int RevCount=0;
int WindDirectionPin=A1;
int DetectPin1=A7;
float AnalogValue=0;
int RawValue=0;
int DirectionNumber = 0;
char Direction[15];
const char DirectionDescription[][15] =
{ "None"
, "North"
, "North East"
, "East"
, "South East"
, "South"
, "South West"
, "West"
, "North West"
, "Err: Batt Low"
};

```

```

//Subroutine to get wind sensor data
//Digital read pin to detect revolutions of wind speed sensor

//Sample time in milliseconds (4000 = 4 seconds)
//Wind speed in in km/hrr
//Since the main program and ISR are sharing this variable
//Analog read pin for wind direction
//Analog pin used to detection the connection status of the wind sensor
//Voltage of the potentiometer from the wind direction sensor

```

```

String WindsensorState="";
byte WindsensorConnected=0;
subroutine
Timer timerWindSpeed(SampleTime1, WindSpeed);
precise timing than milli(s)

//Store the connection status of the wind sensor
//Variable used to control the execution of the wind sensor setup

//Use of timers to calculate wind speed, every 5 seconds. It has more
precise timing than milli(s)

//Subroutines and variables for BME280 operation
void GetBME280SensorData(void);
float Temperature;
float Pressure;
float Humidity;
byte BME280SensorConnected=0;
reconnection to the photon
String BME280SensorState("");
CE_BME280 bme;

//To control the execution of the setup code for the BME280 after its
reconnection to the photon

//To store the connection status of the sensor
//For I2C communication

//Subroutines and variables responsible for battery status monitoring
void BatteryStatus(void);
float BatteryVoltage;
float BatteryVoltageComparison=5;
String BatteryVoltageLow="<3.36";
int BatteryPin=A0;

//Used to prevent battery fluctuation

```

```

signed long ChargeCounter=4501;           //The variable that initiates timed charging and is responsible for the
charging period
double ChargeEndTime=0;                  //The time the battery status must transition to "battery charged"
long ChargeStartTime=0;                  //The time when the battery reaches 4.2V
int STAT1 = D3;
int STAT2 = D4;
WidgetLED ChargeLed(V12);
byte BatteryChargedNotification=1;       //To allow battery charged notification to be executed once.
byte BatteryChargingNotification=1;     //To allow battery charging notification to be executed once.
byte BatteryLowNotification=1;          //To allow battery low notification to be executed once

//Subroutine and variables to obtain and maintaining RTC time
void GetTime(void);
int Day, Month, Year, Hours, Minutes, Seconds;
double t=0;                             //variable used to store the Unix time
double RTCResetTime=43200;              //43200 seconds is equal to 12 hours
double RTCStartTime=1;
double RTCEndTime=1;
byte RTCReset=0;
byte RTCTimeAdjust=1;                   //Controls the execution of calibrating the photon's RTC time after 12
hours have passed
byte AdjustmentAmount=1;                //Reduce the RTC time by 1 second after 12 hours have passed

```

```

//Subroutines and variables for the rain sensor operation
void RainSensor(void);
void Rain(void);
void Tipping(void); //Interrupt subroutine ISR
void RainSensorSetup(void);
int RainDetectorPin=D5;
int RainSensorPin=D6;
int DebounceTime1;
int SampleTime2=900000; //equivalent to 15 minutes
float RainFall=0;
byte RainSensorConnected=0;
String RainSensorState;
volatile int TipCount=0; //Since the main program and ISR are sharing this variable
Timer timer2(SampleTime2, Rain);

// Data logging
void SaveToSDCard(void); //Control data logging
byte DataLogger=0;
byte CheckSDCard=0;
byte SDCount=0;
int DetectPin2=A6;

```

```

void CheckDataLoggerPB(void);                                //Subroutine to maintain the state of the data logging push button on
the app
const int SD_CHIP_SELECT = A2;
SdFat sd;
SdCardPrintHandler printToCard(sd, SD_CHIP_SELECT, SD_SCK_HZ(4 * MHZ));
size_t counter = 0;

//Subroutine responsible for displaying information on the serial monitor
void DisplayOnSerialMonitor(void);

//Subroutine for power save mode
void PowerSaveMode(void);
byte SleepTime=1;                                           //The period, in seconds, that the photon remains in stop mode
int PowerSave1=0;
int PowerSave2=0;

SYSTEM_MODE(MANUAL);                                       //Switches off the Wi-Fi module on the photon upon start up

void setup() {

```

```

Serial.begin(9600);
Serial1.begin(115200);

Blynk.begin(Serial1,auth);

//Set GPIOs as inputs with internal pull/down resistors
pinMode(WindspeedPin,INPUT_PULLUP);
pinMode(DetectPin1, INPUT_PULLDOWN);
pinMode(DetectPin2, INPUT_PULLUP);
pinMode(RainSensorPin,INPUT_PULLDOWN);
pinMode(RainDetectorPin,INPUT_PULLDOWN);

pinMode(STAT1, INPUT_PULLUP);
pinMode(STAT2, INPUT_PULLUP);

Blynk.notify("Weather Station: Online");
//Serial.println("Weather Station: Online");
requestTime();
timer1.setInterval(3000L, AllSubroutines);
seconds

}

```

```

//Initialise serial communication for the serial monitor
//Initialise serial communication on the TX and RX pins with 115200
baudrate for JDY-08 BLE module
//Connect the Blynk application to the photon via Bluetooth

```

```

//Requests Blynk RTC time upon start-up of weather station
//Set timer1 to execute the "AllSubroutines" subroutine every 3

```

```

//Maintains communication between weather station app and hardware
void loop() {

    Blynk.run();                //Maintain connection to Blynk
    timer1.run();               //Keep timer1 running

}

//Executes periodically when timer1 reaches 3 seconds
void AllSubroutines(){

    GetTime();                  //To set and maintain the photon's RTC time
    Windsensor();
    RainSensor();
    GetBME280SensorData();
    //DisplayOnSerialMonitor();
    SaveToSDCard();             //Data logging
    BlynkComms();               //Called last to allow variables to be updated before they are
    transmitted to the mobile app
    PowerSaveMode();            //Checks if stop mode should be activated or not

}

```

```

//Setting up the timer and initialisation of the interrupt responsible for wind sensor measurements
void WindsensorSetup(){

    Speed=0;
    RevCount=0;
    timerWindSpeed.reset();           //The stopped timer will reset and with the next line of code it will
    start                             start
    timerWindSpeed.start();           //Initialize timer for wind speed calculation
    attachInterrupt(WindspeedPin, Revolution, FALLING); //Setup an interrupt that is triggered on the falling edge of the
    signal every time the reed switch closes
    WindsensorConnected=1;
    PowerSave1=0;                     //Switch sleep mode off

}

//Primary wind sensor subroutine
void Windsensor(){

    if(digitalRead(DetectPin1) == HIGH){ //To detect if the wind sensor is connected to the weather station

        if (WindsensorConnected == 0){ //Setup wind sensor only the first time after it is detected

            WindsensorSetup();
        }
    }
}

```



```

    WindsensorState="Connected";
    //Serial.println("Windsensor is connected and setup is complete");

}

if(BatteryVoltage<=3.36){

    DirectionNumber=9;

}
else{

    if(Speed==0){                                     //If there is no wind speed then the wind direction must be blocked

        DirectionNumber=0;

    }
    else{

        WindDirection();                             //Determine wind direction

    }
}
}

```

```

else if(digitalRead(DetectPin1) == LOW){                                //If the sensor is not connected disable the timer and interrupt once

    timerWindSpeed.stop();
    detachInterrupt(WindspeedPin);
    WindsensorConnected=0;
    WindsensorState="Disconnected";
    RevCount=0;
    Speed=0;
    DirectionNumber=0;
    //Serial.println("Windsensor is not connected");
    PowerSave1=1;

}

strcpy(Direction, DirectionDescription[DirectionNumber]); //Store the wind direction measurement into variable "direction"
//Serial.printlnf("Wind direction is: %s", Direction);

}

//Interrupt driven subroutine that executes when a pulse is detected
void Revolution(){

    //Software debounce solves reed switch false readings.

```

```

    if(millis()-DebounceTime > 15){

        RevCount=RevCount+1;
        DebounceTime=millis();

    }

}

//Obtain wind speed
void WindSpeed(){

    Speed=(3.621015*RevCount)/(SampleTime1/1000);           //Speed in km/hrr. Formula adapted from technical manual check
    bookmarks Davis says 1600 rev/hr ---> 1 mile/hour
    RevCount=0;
    timerWindSpeed.reset();                                //Resets the timer to start from 0

}

//Obtaining wind direction
void WindDirection(){

    RawValue = analogRead(WindDirectionPin);
    DirectionNumber = map(RawValue, 0, 4096, 1, 9);

```

```
}
```

```
//Temperature, humidity and pressure function
```

```
void GetBME280SensorData(){
```

```
    //Check if the BME280 sensor is connected to the weather station
```

```
    if(!bme.begin(Addr)){                                //BME280 sensor is disconnected
```

```
        BME280SensorState="Disconnected";
```

```
        BME280SensorConnected=0;                        //To prevent this 'if' statement from
```

```
        Executing continuously
```

```
        //Serial.println("\nBME280 sensor not connected");
```

```
        //Since sensor is disconnected, zero all measurements to prevent false readings
```

```
        Temperature = 0;
```

```
        Humidity = 0;
```

```
        Pressure = 0;
```

```
    }
```

```
    else if(bme.begin(Addr)){                            //BME280 sensor is connected
```

```
        //Allow the sensors readings to stabilise before considering the measurements
```

```

if(BME280SensorConnected==0){

    BME280SensorState="Initialising...";           //To give the BME280 time to produce stable readings
    BME280SensorConnected=1                       //To prevent this 'if' statement from executing continuously
    //Serial.println("BME280 sensor is initialising...");

}
else if(BME280SensorConnected==1){

    BME280SensorState="Connected";
    BME280SensorConnected=2;                       //To prevent this 'if' statement from executing continuously
    //Serial.println("BME280 sensor is connected and setup is complete");

}
if(BME280SensorConnected==2){

    delay(30);                                     //To allow BME280 sensor time to initialise
    Pressure = (bme.readPressure()/100);           //Pressure measurements in hPa
    Temperature = bme.readTemperature();
    Humidity = bme.readHumidity();

}
}

```

```
}
```

```
//Primary rain sensor subroutine
```

```
void RainSensor(){
```

```
    if(digitalRead(RainDetectorPin)==HIGH){
```

```
        if(RainSensorConnected==0){
```

```
            RainSensorSetup();
```

```
            //Serial.println("Rain sensor is connected and setup is complete");
```

```
            RainSensorState="Connected";
```

```
        }
```

```
    }
```

```
    //If the rain sensor is not connected disable the interrupt and timer
```

```
    else{
```

```
        detachInterrupt(RainSensorPin);
```

```
        timer2.stop();
```

```
        TipCount=0;
```

```
        RainFall=0;
```

```

    RainSensorConnected=0;
    RainSensorState="Disconnected";
    //Serial.println("Rain sensor is not connected");
    PowerSave2=1;
}

}

//Rain sensor setup subroutine
void RainSensorSetup(){

    TipCount=0;
    attachInterrupt(RainSensorPin, Tipping, RISING);
    timer2.reset();
    timer2.start();
    RainSensorConnected=1;
    PowerSave2=0;

}

//Interrupt driven subroutine that executes when a pulse is detected
void Tipping(){

    //Software debounce

```

```

    if(millis()-DebounceTime1 >15){

        TipCount=TipCount+1;
        DebounceTime1=millis();

    }
}

//Executes every 15 minutes to obtain rainfall measurements
void Rain(){

    RainFall=((TipCount*0.2)*(3600000/SampleTime2)); //Rainfall in mm/hr
    TipCount=0;
    timer2.reset();
    timer2.start();

}

//Responsible for sending weather, sensor connection status and battery status data to the Blynk mobile application
void BlynkComms(){

    //BME280 sensor data
    Blynk.virtualWrite(V0, Temperature);
    Blynk.virtualWrite(V1, Humidity);

```



```

Blynk.virtualWrite(V2, Pressure);
Blynk.virtualWrite(V8, BME280SensorState);           //Sensor connection status

//Wind sensor data
Blynk.virtualWrite(V5, Direction);
Blynk.virtualWrite(V6, Speed);
Blynk.virtualWrite(V9, WindsensorState);

//Rainfall sensor data
Blynk.virtualWrite(V15, RainFall);
Blynk.virtualWrite(V16, RainSensorState);

//Battery monitoring subroutine
BatteryStatus();

}

void BatteryStatus(){

    BatteryVoltage=((analogRead(BatteryPin)*4.2)/3510))+0.015;           //Calibrated battery voltage

    //Non charge related battery conditions
    if((digitalRead(STAT1)==HIGH && digitalRead(STAT2)==HIGH) || (digitalRead(STAT1)==LOW && digitalRead(STAT2)==LOW)){

```

```
Blynk.setProperty(V13, "label", "Battery voltage");  
ChargeCounter=4501;
```

```
BatteryChargedNotification=1;  
BatteryChargingNotification=1;
```

```
if(BatteryVoltage <= BatteryVoltageComparison){  
  
    BatteryVoltageComparison=BatteryVoltage;  
    //Serial.println("BatteryVoltageComparison=BatteryVoltage");  
  
}  
else if(BatteryVoltage > BatteryVoltageComparison){  
  
    BatteryVoltage=BatteryVoltageComparison;  
    //Serial.println("BatteryVoltage=BatteryVoltageComparison");  
  
}  
  
//Check battery voltage  
if(BatteryVoltage<=3.36){
```

```

    Blynk.setProperty(V12, "color", "#F00606");           //Red LED
    Blynk.setProperty(V12, "label", "Battery low");         //Set the widget to display that the battery is low
    //Serial.println("Battery critically low. Please connect charger immediately!");
    Blynk.notify("Battery critically low. Please connect charger immediately!");

}
else if(BatteryVoltage<=3.4 && BatteryLowNotification==1){ //Low voltage condition at 3.4V

    Blynk.setProperty(V12, "color", "#F00606");           //Red LED
    Blynk.setProperty(V12, "label", "Battery low");         //Set the widget to display that the battery is low
    //Serial.println("Battery is low. Charge battery");
    Blynk.notify("Battery is Low. Please connect charger!");
    BatteryLowNotification=0;

}
else if(BatteryVoltage>3.40){

    Blynk.setProperty(V12, "color", "#23C48E");           //Green LED
    Blynk.setProperty(V12, "label", "Battery normal");       //Set the widget to display that the battery is
normal
    //Serial.println("Battery is Normal");

}

```

```

}
if(digitalRead(STAT1)==LOW && digitalRead(STAT2)==HIGH && ChargeCounter>0){ //Battery charging condition

    BatteryVoltageComparison=5; //To ensure that this variable will be set
    with the new battery voltage under normal battery operating conditions
    BatteryChargedNotification=1;
    BatteryLowNotification=1;

    if(BatteryChargingNotification==1){

        Blynk.notify("Battery charging");
        BatteryChargingNotification=0;

    }

    Blynk.setProperty(V12, "color", "#DFED00"); //Yellow LED
    Blynk.setProperty(V12, "label", "Battery charging"); //Set the widget to display that the battery is
    charging
    Blynk.setProperty(V13, "label", "Battery charge voltage");

    if(BatteryVoltage>=4.2 && (PowerSave1==0 || PowerSave2==0)){ //To ensure that the changing of the charge status by
    timing is only initiated when the power saving feature is not active

```

```

t=Time.now();
    photon's RTC

//Setting "t" to be the current time from the

if(ChargeCounter==4501){

    ChargeStartTime=t;
    ChargeCounter=4500;
    minutes.
    ChargeEndTime=ChargeStartTime+ChargeCounter;

//4500 seconds is equivalent to 1 hour and 15
//Determine the end of the charging period

}
else{

    if(ChargeEndTime<=t){

        ChargeCounter=-1;

    }

}

}

```

```

}
if((digitalRead(STAT1)==HIGH && digitalRead(STAT2)==LOW) || (ChargeCounter<=0 && digitalRead(STAT1)==LOW &&
digitalRead(STAT2)==HIGH)){      // Battery charged conditions

    ////Resets the "ChargeCounter" if fully charged state was achieved by STAT pins
    if(digitalRead(STAT1)==HIGH && digitalRead(STAT2)==LOW){

        ChargeCounter=4501;

    }

    Blynk.setProperty(V12, "color", "#1A6BF4");           //Blue LED on
    Blynk.setProperty(V12, "label", "Battery charged");    //Set the widget to state that the battery is charged
    Blynk.setProperty(V13, "label", "Battery voltage");
    //Serial.println("Battery is Charged");

    BatteryChargingNotification=1;

    //Executes once
    if(BatteryChargedNotification==1){

        Blynk.notify("Battery charged");
        BatteryChargedNotification=0;
    }
}

```

```

    }

}
//Updates the battery voltage on the display widget
if(BatteryVoltage<=3.36){

    Blynk.virtualWrite(V13, BatteryVoltageLow);           //Display "<3.36V" in battery voltage display widget
    //Serial.println("Battery voltage is <3.36V");

}
else{

    Blynk.virtualWrite(V13, BatteryVoltage);
    //Serial.printf("Battery voltage is %0.3f", BatteryVoltage);

}

}

//Executes when the data logger PB is toggled on the app
BLYNK_WRITE(V14){

    DataLogger = param.asInt();                           //The state of the PB

```

```

//Activate data logging
if(DataLogger==HIGH){

    CheckSDCard=1;

}
else{

    Blynk.notify("Data logger: Disabled");
    //Serial.println("Save data to SD card: Disabled");

}

}

//Primary subroutine that enables/disables data logging to the SD card
void SaveToSDCard(){

    //Execute when the data logger feature is activated and the battery voltage is not critically low
    if(DataLogger==1 && BatteryVoltage>3.36){

        //Checks if the SD card is connected

```



```

if (digitalRead(DetectPin2)==HIGH){

    //Separate the data batches stored on the SD card
    if(CheckSDCard==1){

        printToCard.println("\nNew batch of weather data,Date,Time,Temperature(C),Humidity(%),Pressure(hPa),Wind
        speed(km/hr),Wind direction,Rain fall(mm/hrr)");    //This will run once only when the SD is first activated
        //Serial.println("\nNew batch of weather data");

        if(!printToCard.getLastBeginResult()){

            Blynk.notify("Data logger initializing, please wait...");
            Serial.println("Data logger initializing, please wait...");
            CheckSDCard=1;
            SDCount++;

            if(SDCount==10){                                //If the data logger fails after 10 attempts then a fault
            indication is produced

                Blynk.notify("Data logger failed");
                Blynk.virtualWrite(V14,0);                    //Switch PB state to "Off"
                DataLogger=0;                                //Disable the data logger
                SDCount=0;
        }
    }
}

```

```

    }
}
else{

    Blynk.notify("Data logger: Enabled");
    Serial.println("Data logger: Enabled");
    CheckSDCard=0;
    SDCount=0;

}
}
else if(CheckSDCard==0){                                     //Log timestamped data to SD card

    t=Time.now();
    Seconds=Time.second(t);
    Minutes=Time.minute(t);
    Hours=Time.hour(t);
    Day=Time.day(t);
    Month=Time.month(t);
    Year=Time.year(t);

    printToCard.printf(" ,%i/%i/%i,%i:%i:%i,%0.2f,%0.2f,%0.2f,%0.2f,%s,%0.2f", Day, Month, Year, Hours, Minutes,
    Seconds, Temperature, Humidity, Pressure, Speed, Direction, RainFall);

```

```

        //Serial.println("%i/%i/%i %i:%i:%i Temperature: %0.2f C, Humidity: %0.2f %, Pressure: %0.2f hPa, Windspeed:
        %0.2f km/hr, Wind direction: %s, Rainfall: %0.2f mm", Day, Month, Year, Hours, Minutes, Seconds, Temperature,
        Humidity, Pressure, Speed, Direction, RainFall);

    }
}

//Executes if the data logging is active/activated but no SD card is present in the SD card reader
else{

    //Serial.println("No SD card present");
    Blynk.notify("No SD card present");
    Blynk.virtualWrite(V14,0);
    DataLogger=0;
    SDCount=0;

}
}

//Disable data logger from activating when battery is critically low
else if(DataLogger==1 && BatteryVoltage<=3.36){

    printToCard.println("Data logger de-activated. Battery is critically low. ");
    Blynk.notify("Data logger de-activated. Battery is critically low. ");
    Blynk.virtualWrite(V14,0);

```

```

    DataLogger=0;

}

}

//Checks if sleep mode should be enabled
void PowerSaveMode(){

    if(PowerSave1==1 & PowerSave2==1){           //Power saving mode activated when both the rain sensor and the
    wind sensor are disconnected from the station

        System.sleep(DetectPin1,RISING,SleepTime);
        CheckDataLoggerPB();                       //Update the state of the data logger PB after returning from stop mode

    }

}

//obtain, set and maintain the photon's RTC time
void GetTime(){

```

```

if(RTCReset==0 && RTCEndTime<=Time.now()){

    requestTime();

    if(float(((Time.now()-RTCStartTime)/RTCTimeAdjust))>=RTCResetTime){ //The Photon's RTC is corrected every 12 hours

        t=(Time.now()-AdjustmentAmount); //Every 12 hours the time is adjusted, ONCE, by 1
        second
        Time.setTime(t);
        RTCTimeAdjust++;

    }

}

if(RTCReset==1 && RTCEndTime<=Time.now()){ //Executes once only, after the photon's RTC is
reset by Blynk

    RTCEndTime=RTCStartTime+RTCResetTime; //The time at which the photon's RTC needs to be
reset by Blynk
    RTCReset=0;

}

}

```

```

//Requests Blynk Unix time
void requestTime(){

    Blynk.sendInternal("rtc", "sync");

}

//Obtaining and converting Unix time to normal date and time format
BLYNK_WRITE(InternalPinRTC) {

    t = param.asLong();                //Unix time returned by Blynk
    RTCStartTime=t;
    Time.setTime(t);                   //Set the photon's RTC according to Blynk's RTC time

    RTCReset=1;                        //Controls the setting of the RTCEndTime
    //Serial.println("Photon set to Blynk RTC time");
    RTCTimeAdjust=1;                   //Used to adjust the photon's RTC time when 12 hours of operation is
    reached

}

//Executes when the weather station hardware connects to the app

```

```

BLYNK_CONNECTED(){

    Blynk.notify("Weather Station: Connected");
    //Serial.println("Weather Station: Connected");

    CheckDataLoggerPB(); //Update the state of the data logger PB on the mobile app

    //Ensures that the battery status notifications occur upon reconnection to the app so that they are not missed
    BatteryChargedNotification=1;
    BatteryChargingNotification=1;
    BatteryLowNotification=1;

}

//Checks the last known state of the data logger PB and updates it on the app
void CheckDataLoggerPB(){

    if(DataLogger==0){

        Blynk.virtualWrite(V14,0);

    }
    else if(DataLogger==1){

```

```

        Blynk.virtualWrite(V14,1);

    }

}

//Displays all the sensor and connection status data to the serial monitor for debugging purposes
void DisplayOnSerialMonitor(){

    Serial.println("Temperature %f C", Temperature);
    Serial.printf("Humidity %f", Humidity);
    Serial.print(" % ");
    Serial.println("\nPressure %f hPa", Pressure);
    Serial.println("Rev Count: %d", RevCount);
    Serial.println("Windspeed %f km/hr", Speed);
    Serial.println("Analog voltage: %fV", AnalogValue);
    Serial.print("Wind direction: ");
    Serial.print(Direction);
    Serial.print("\nTemperature, pressure and humidity sensor are ");
    Serial.print(BME280SensorState);
    Serial.println("");
    Serial.print("Wind speed and direction sensors are ");
    Serial.print(WindsensorState);

```



```
Serial.println("");  
Serial.print("Rain fall sensor is ");  
Serial.print(RainSensorState);  
Serial.println("");  
  
}
```