

Clemson University

TigerPrints

All Theses

Theses

8-2022

Miniaturized Battery Powered Air Quality Monitoring System

Bryan Chacon

bchacon@clemson.edu

Follow this and additional works at: https://tigerprints.clemson.edu/all_theses



Part of the [Biomedical Commons](#), and the [Electrical and Electronics Commons](#)

Recommended Citation

Chacon, Bryan, "Miniaturized Battery Powered Air Quality Monitoring System" (2022). *All Theses*. 3863.
https://tigerprints.clemson.edu/all_theses/3863

This Thesis is brought to you for free and open access by the Theses at TigerPrints. It has been accepted for inclusion in All Theses by an authorized administrator of TigerPrints. For more information, please contact kokeefe@clemson.edu.

Miniaturized Battery Powered Air Quality Monitoring System

A Thesis
Presented to
the Graduate School of
Clemson University

In Partial Fulfillment
of the Requirements for the Degree
Master of Science
Electrical Engineering

by
Bryan Anthony Chacon
August 2022

Accepted by:
Dr.Goutam Koley, Committee Chair
Dr.William Harrell
Dr.Judson Ryckman

ABSTRACT

In this work, an air quality monitoring system was developed using various sensors that measure specific air quality parameters, including Volatile Organic Compounds, Carbon Dioxide, particulate matter of varying sizes, ambient pressure, humidity, and temperature. This system is based off a Particle micro-controller, Boron LTE CAT-M1 which allows for cellular connectivity for real-time data transmission. It is powered by a 3.7 Volt Li-Po battery and has a miniaturized design which allows for portability. This data is processed through an Internet of Things software provider that allows for the device to be connected and accessed to and from anywhere in the world. This paper discusses the design considerations, prototyping phase, electrical circuit design phase, printed circuit board design phase and fabrication process phase information. This paper also compares the performance of the air quality monitoring device to previous iterations and existing commercial devices.

DEDICATION

This work is dedicated to my wonderful family, friends, and partner for all their support and encouragement during this process.

ACKNOWLEDGMENTS

I would like to acknowledge my advisory committee members, Dr. Goutam Koley, Dr. William Harrell, and Dr. Judson Ryckman, consideration of my work. I would like to recognize my advisor, Dr. Goutam Koley, for his guidance as I completed my Graduate course of study.

I would also like to thank fellow graduate student Balaadithya Uppalapati for his input and assistance within the NESL lab. I would like to also express my appreciation for some undergraduate students for their assistance in software development of the device including Kevin Patel and Dhruv Acharya.

Finally, I would like to acknowledge my mother, father, sister, brother and partner for their love and support throughout this journey which started during the global pandemic caused by Coronavirus. I could not have persevered and reached this point in my academic/professional career without them.

TABLE OF CONTENTS

	Page
TITLE PAGE	i
ABSTRACT	ii
DEDICATION	iii
ACKNOWLEDGMENTS	iv
LIST OF TABLES	viii
LIST OF FIGURES	x
 CHAPTER	
1. INTRODUCTION	1
1.1 The Need for Air Quality Monitoring.....	1
1.2 Air Quality Parameters	2
2. DESIGN OF THE SENSOR SYSTEM	4
2.1 Sensors and Components	4
2.2 Integration of Sensors and Components	10
3. BREADBOARD BASED DEMONSTRATION	11
3.1 Individual Component Circuit Diagrams.....	11
3.2 Breadboard Based Prototype Circuit	16
4. SOFTWARE INTEGRATION AND TESTING.....	18
4.1 Component Testing and Software Development	18
4.1.1 Microcontroller Setup	18
4.1.3 THP Sensor Setup	19
4.1.4 VOC Setup	20
 Table of Contents (Continued)	

Page

4.1.5 Particulate Matter Sensor Setup	20
4.1.6 Carbon Dioxide Sensor Setup	21
4.1.7 Global Positioning System Sensor Setup	21
4.2 Combined Sensor Component Testing	29
5. SENSOR DATA DASHBOARD DEVELOPMENT.....	35
5.1 Data Stream Creation and Handling	35
5.2 Losant Website and Data Processing.....	37
5.3 Data Visualization and User Interface	40
6. FINAL CIRCUIT AND PCB DESIGN.....	41
6.1 Sensor Schematic Designs	41
6.2 Final Circuit Schematic Design	47
6.3 Printed Circuit Board Design.....	48
6.4 Printed Circuit Board Fabrication Process.....	53
7. SUMMARY AND FUTURE PLANS	54
7.1 Summary of Work.....	54
7.2 Possible Future Updates.....	55
APPENDICES	56
A: BME280 Software	58
B: SCD30 Software	59
C: SPS30 Software	60
D: VOC Software.....	61
E: GPS Software.....	62
F: Full System Software.....	64

Table of Contents (Continued)

	Page
H: Completed Software.....	71

J: Microprocessor Data Sheet	77
K: Sensor Data Sheets.....	78
REFERENCES	79

LIST OF TABLES

Table	Page
1.1 Air Quality Parameter Ranges	3
2.1 Particle Boron Specifications.....	5
2.2 SCD30 CO ₂ Sensor Specifications	6
2.3 SPS30 Particulate Matter Sensor Specifications.....	7
2.4 SGP30 VOC Sensor Specifications	8
2.5 XA1110 GPS Sensor Specifications.....	8
2.6 BME280 THP Sensor Specifications.....	9
3.1 Sensor Interface Protocols	12
3.2 BME280 Breakout Board Pin Out Diagram	16
3.3 SGP30 Breakout Board Pinout Diagram	16
3.4 SPARKFUN GPS Breakout Board Pinout Diagram	16
3.5 SCD30 Pin Out Diagram	17
3.6 SPS30 Pin Out Diagram.....	17
4.1 First System Test Resultant Graphs with Impulse Inputs.....	31
4.2 Mobile Full System Test GPS Plots	32
4.3 Mobile Full System Test GPS Maps.....	33

List of Tables (Continued)

Table	Page
6.1 Final System Current Draw and Power Consumption.....	47

6.2	Bill of Materials for System.....	52
-----	-----------------------------------	----

LIST OF FIGURES

Figure		Page
1.1	Poor Air Quality Effects on the Human Body	2
3.2	Particle Electron Pin Connections	14
3.3	Breadboard-based Prototype Circuit Diagram.....	17
3.1	System Block Diagram	11
3.2	UART Communication Diagram.....	12
3.3	SPI Communication Diagram.....	13
3.4	I2C Communication Diagram.....	14
3.5	Particle Boron Pinout Diagram.....	15
3.6	Pull up Resistor I2C Bus Topology	18
3.7	Breadboard Prototype Circuit Diagram	21
5.1	Losant Data Workflow.....	36
5.2	Device State Layer	37
5.3	Data Path Block Diagram	37
5.4	Losant Data Visualization Dashboard	38
6.1	Boron Schematic Symbol with corresponding PCB Footprint.....	39
6.2	BME280 Schematic Symbol with corresponding PCB Footprint	40
6.3	Recommended I2C connection circuit for BME280	41
6.4	Electrical Circuit Schematic for BME280	41

List of Figures (Continued)

Figure	Page
6.5 Recommended Configuration for SGP30	42
6.6 Electrical Circuit Schematic for SGP30	42
6.7 SCD30 Electrical Circuit Schematic.....	43
6.8 5V Boost Circuit Schematic for SPS30 Input.....	44
6.9 Logic Level Shifting Circuit Schematic	45
6.10 GPS Electrical Circuit Schematic	45
6.11 Electrical Circuit Schematic of Full System.....	46
6.12 All components and connections in PCB software.....	48
6.13 PCB Component placement with connections.....	49
6.14 Printed Circuit Board Design for Full System.....	50
6.15 Printed Circuit Board Gerber Files Preview	51

CHAPTER ONE

INTRODUCTION

1.1 The Need for Air Quality Monitoring

Poor air quality is predominately produced/emitted by human activities and are a major threat to mankind and the environment. It is important that we can sense our surroundings and capture the data so that the correct precautionary and safety actions can be executed. Poor air quality has been linked to numerous acute and chronic conditions ranging from decreased concentration, decreased cognitive function, sleep disorders, and/or autoimmune conditions like asthmas, cancer, or nervous system damage. Poor Air Quality can not only cause new health conditions, but it can exacerbate existing health conditions that are common to varying groups which could lead to frequent hospitalization and sometimes death. Studies show that about 16% of Americans alone have been diagnosed or undiagnosed with asthma, chronic obstructive pulmonary disease (COPD), and other respiratory conditions. According to the World Health Organization (WHO) up to 91% of the world's population live in areas where air pollution exceeds safe daily limits. [1] These poor air quality conditions are the fourth largest killer of people on Earth with 7 million people dying each year due to constant exposure and inhaling of these poor air quality conditions [1]. The most dangerous air pollutants that are a root cause behind such terrible statistics are not visible and odorless to humans. Carbon Dioxide (CO₂) and Volatile Organic Compounds (VOCs) are one of the most dangerous air pollutants that humans can be exposed to on a normal day. VOCs are

human made chemicals that can be found in the manufacturing of paint, pharmaceuticals, and refrigerants. VOCs can be emitted by normal activities around the house including cooking, cleaning, and/or painting. CO₂ can be found in the emissions from human activities such as burning coal, oil, and gas. VOC and CO₂ concentrations are higher indoors and it is important that we can detect these harmful air pollutants to protect ourselves and the environment. A summary of some of the affects poor air quality can be seen in the Figure 1.1[12] below.

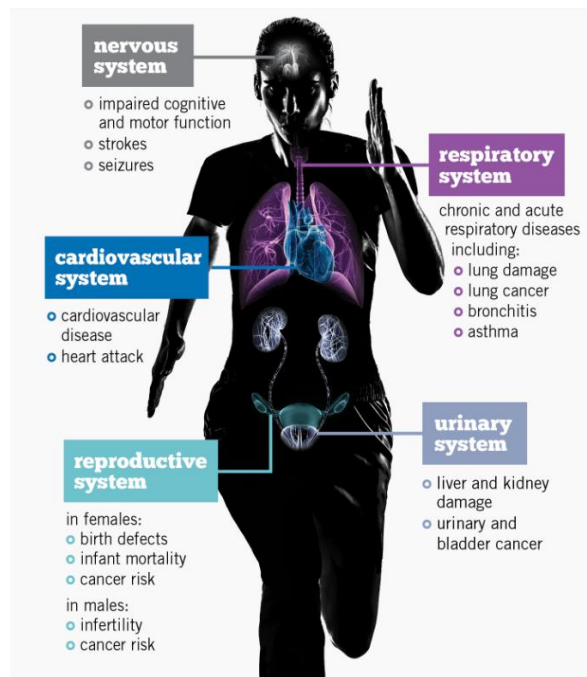


Figure 1.1 Poor Air Quality Effects on the Human Body

1.2 Air Quality Parameters

When determining Air Quality, it is important to include the pollution levels of other important human health factors such as temperature, humidity, and particulate matter which is a mixture of solid particles and liquid droplets found in the air (i.e., dust, dirt, soot, or smoke). The system requirements for our device include the ability to

measure CO₂, VOCs, particulate matter of multiple different sizes, temperature, pressure, and humidity. Air Quality ratings range from excellent, fine, moderate, poor, very poor, and severe. For our system, we will be monitoring all these parameters and sending the real time data which will fit within these ranges for consumer information. It is recommended that the indoor air temperature should range from 73°F to 79°F in the summer and 68°F to 75°F in the winter [2]. High humidity levels (greater than 60%) can lead to mold growth and possibly dehydration which can be dangerous [3]. High CO₂ and VOC levels can lead to lowered cognitive function, drowsiness, and lower activity levels. Furthermore, prolonged CO₂ concentrations above 1,000 ppm generally indicate inadequate ventilation [4]. A table summarizing the various levels of indoor and outdoor levels of the various parameters is given in Table 1.1. All these parameters and ranges are directly used to determine the Air Quality and will give accurate and informative information to the consumer.

Table 1.1 Air Quality Parameter Ranges

Parameter	Outdoor	Indoor (norm)	Indoor (high)	Dangerous
CO ₂ (ppm) [5]	250-350	350-1,000	1,000-5,000	>40,000
VOC (ppb) [5]	0-10	0-220	220-2200	>2200
Particulate Matter (ug/m ³) < PM2.5 [5]	<1	<12	20-60	>60
Particulate Matter (ug/m ³) < PM10 [5]	<1	<20	50-100	>100
Temperature (F) [6]	70-85	70-82	90-130	>130

CHAPTER TWO


SENSOR COMPONENT SELECTION

2.1 Sensors and Components

The components and sensors used in our system were selected based on their sensitivity, selectivity, size, response time, and power consumption. For our system, we have decided to use a Boron Particle LTE CAT-M1, SPS30 sensor, SGP30 sensor, SCD30 sensor, BME280 sensor, XA1110 sensor as the microcontroller, particle matter sensor, Volatile Organic Compound sensor, CO2 sensor, Temperature Pressure Humidity sensor, and Global Position System sensor, respectively.


The Boron Particle is a develop kit that supports cellular networks and Bluetooth LE (BLE). It is based on an ARM Cortex-M4F 32-Bit processor, Nordic nRF52840 which is the Bluetooth module and has a built-in battery connector and charging circuit which allows for a Lithium Polymer (Li-Po) battery to be connected and power the components and sensors within our system. Particle's IoT connectivity services and microcontrollers made this selection uncomplicated. Table 2.1 shows a some of the Particle Boron key specifications that are related to our needs.

Table 2.1 Particle Boron Specifications

<p>Microcontroller</p>	
<p>Specification</p>	<p>Particle</p>
<p>Controller</p>	<p>Boron</p>
<p>Size (inches)</p>	<p>0.9x0.65x2.00</p>
<p>Weight</p>	<p>10 grams</p>
<p>Cost</p>	<p>\$59.37</p>
<p>Input Voltage</p>	<p>3.3-6.5 V</p>
<p>Digital Pins</p>	<p>20</p>
<p>Analog Inputs</p>	<p>6</p>
<p>Analog Outputs</p>	<p>6</p>
<p>PWM Outputs</p>	<p>8</p>
<p>UART Ports</p>	<p>3</p>
<p>SPI Ports</p>	<p>1</p>
<p>I2C Ports</p>	<p>2</p>
<p>JTAG Ports</p>	<p>1</p>
<p>Built-in Wi-Fi</p>	<p>No</p>
<p>Built-in Cellular</p>	<p>Yes</p>
<p>Built-in Bluetooth</p>	<p>Yes</p>

The Sensirion SCD30 (SCD30) is the Carbon Dioxide sensor in our system. It is a nondispersive infrared sensor which means that it detects the decrease in transmitted infrared light which is in proportion to the gas concentration. The CO₂ particles absorb infrared light and create vibrations for the sensor to detect. The black box component on the breakout board is where the infrared light is constantly being sent and waits for CO₂ particles to create the reaction described above. The SCD30 needs to be powered by a 3.3Volt-5Volt source and can use UART or I2C for communication. An overview of the specification of the SCD30 are shown in Table 2.2.


Table 2.2 CO₂ Sensor Specifications

Sensor	
Sensor	SCD30
Company	Sensirion
Price	\$58
Voltage	3.3–5.5 V
Peak Current	75 mA
Lifetime	15 years
Range	400-10,000 ppm
Accuracy	30 ppm
Response Time	20 s
Interface	UART / I2C
Dimensions	(35x23) mm / (1.4x1.0) inches
Footprint	805mm ² / 1.25inches ²
Other Qualities	Measures temperature and humidity

The Sensirion SPS30 sensor is our particulate matter sensor. It is based on laser scattering and Sensirion’s latest contamination-resistance technology. This sensor can


sense 4 different sizes ranging from PM1.0, PM2.5, PM4, and PM10, where the number stands for the range in microns. The SPS30 needs a 5 Volt supply and can communicate via I2C and UART. The SPS30 specifications can be seen in Table 2.3.

Table 2.3 SPS30 Specifications

Sensor	
Sensor	SPS30
Company	Sensirion
Price	\$52
Voltage	4.5-5.5 V
Peak Current	80 mA
Lifetime	>8 years
Particle Size	1, 2.5, 4, 10
Range (mass)	0-1000 $\mu\text{g}/\text{m}^3$
Range (concentration)	0-3000 $\#/ \text{cm}^3$
Accuracy	10
Response Time	5 s
Interface	UART / I2C
Dimensions	(41x41) mm / (1.6x1.6) inches
Height	12 mm / 0.5 inches
Other Qualities	Mass and Concentration


The SGP30 Sensor from Sensirion will detect the Volatile Organic Compounds (VOCs). The SGP30 sensor is a standard hot-plate MOX sensor, this means it is composed of a metal-oxide surface (often tin dioxide), a sensing chip that measures the change in conductivity. The SGP30 requires a 1.62V-1.98Volt power supply and uses I2C communication protocol. The SGP30 specifications can be seen in Table 2.4.

Table 2.4 SGP30 Specifications

Sensor	
Sensor	SGP30
Company	Sensirion
Price	\$13.49
Voltage	1.62-1.98 V
Range	0-1000 ppm
Interface	I2C
Dimensions	2.45 x 2.45 mm
Height	0.9 mm

We used the XA1110 module from GTOP as our GPS sensor for the device. The XA1110 is supported by several GPS constellations, which means different Global Position System architectures in space that operate in case one fails. This allows for accurate and reliable tracking. It is powered by a 3.3 Volt source, has a connector to connect an external antenna and uses I2C communication protocol. The XA1110 specifications can be seen in Table 2.5.

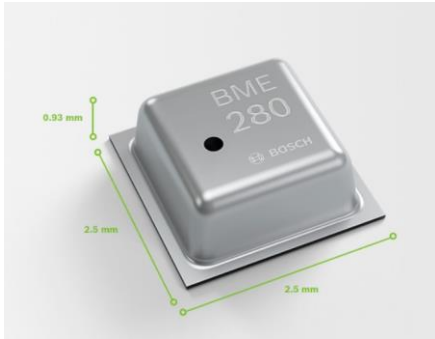
Table 2.5 XA1110 Specifications

GPS Sensor	
Sensor	XA1110
Company	Sierra Wireless
Antenna Location	Internal and/or External

Price	\$20.85
Voltage	3.3 V
Peak Current	20-35 mA (25 mA typical)
Sensitivity	-165 dB
Interface	UART or I2C
Dimensions	12.5 mm x 12.5 mm
Height	6.8 mm

Finally, the BME280 from Bosch will be used to detect Temperature, humidity, and pressure. This IC uses common practices to sense the intended parameters. It is small, robust, has a fast response time, and requires a low amount of power to operate. BME280 specifications can be shown in Table 2.6.

Table 2.6 BME280 Sensor Specifications

	Voltage	1.7-3.6 V	
	Peak Current	< 800 μ A	
	Price	\$8	
	Temperature	Humidity	Pressure
Range	-40-185°F	0-100 %RH	0.3-1.1 atm
Accuracy	1°C / 1.8°F	3	1.2x10 ⁻⁴ atm
Resolution	0.01°C / 0.018°F	0.008	1.776x10 ⁻⁶ atm
Response Time	1s	Dimensions	2.5 x 2.5 x 0.93 mm
Interface	I2C / SPI		

2.2 Integration of Sensors and Components

After selecting all components that will be fused together to sense the intended parameters, a plan to test and design the overall system was devised. To do so, breakout boards from trusted electronic suppliers, SPARKFUN and ADAFRUIT were acquired.

When prototyping IoT devices that fuse multiple sensors, using existing breakout boards to create a breadboard-based circuit with all sensors is an intelligent step to verify if all the sensors chosen will/can work together. Breakout boards are individual printed circuit boards that have the desired sensing Integrated Chips with their own corresponding power regulating and communication circuits designed from the electronic suppliers, that then allow the data transfer to your intended Microcontroller. The Particle Boron's ability to transfer data to and from a cloud service and ability to host over 100 different devices on its Serial Communication Line and Serial Data lines make it a sound choice to test with our 5 sensors/breakout boards and allow for additions in the future if intended.

CHAPTER THREE

BREADBOARD BASED CIRCUIT DESIGN

3.1 Individual Component Circuit Diagrams

The next step in the design process was to acquire the necessary breakout boards to create a circuit to be built on a standard bread board. This step is important because different sensors and Serial Communication and Serial Data Lines require different impedances and voltages to be able to communicate. Before creating the circuit design, it is important to understand the overall process flow of our IoT device. Figure 3.1 shows the process flow for our IoT device to execute.

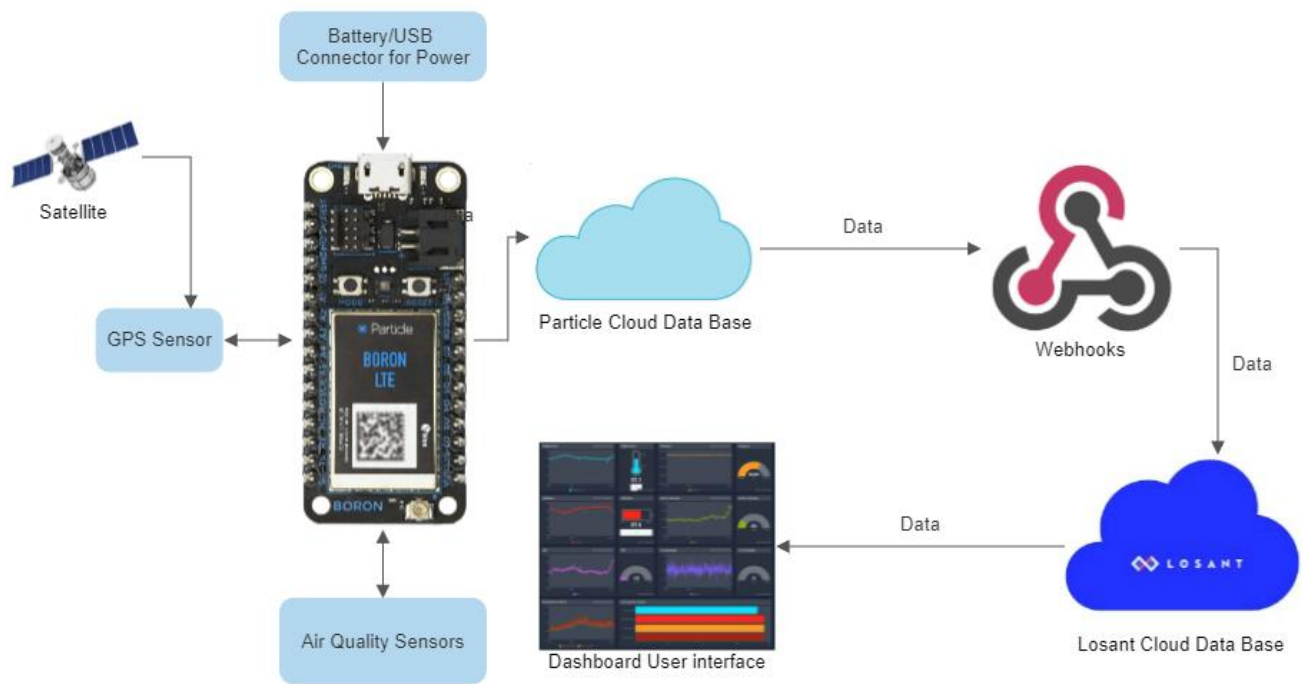


Figure 3.1 System Block Diagram

To create the circuit for our system, we need to decide how all the breakout boards will be powered and how they will communicate with our Microcontroller, the Particle Boron. A communication protocol is a system of rules that allow for two or more devices with communication systems to transfer information via the corresponding physical quantity, in our case the data from the sensing mechanisms of each sensor. The communication protocols supported by each sensor in our system can be seen in Table 3.1.

Table 3.1 Sensor Interface Protocols

Supported Protocols	BME280 (ATM)	SCD30 (CO ₂)	SPS30 (PM)	SGP30 (VOC)	XA1110 (GPS)
UART		✓	✓		✓
SPI	✓				
I2C	✓	✓	✓	✓	✓

The most common communication protocols that are in use today are Universal Asynchronous Receiver Transmitter (UART), Serial Peripheral Interface (SPI), and Inter-Integrated Circuit Communication(I2C). UART communication protocol has two data lines, one that transmits data (TX) and one that receives data (RX) which can be seen as digital I/O pins in microcontrollers. The TX and RX of the chosen microcontroller will communicate to the desired sensor as shown in Figure 3.2 [9].

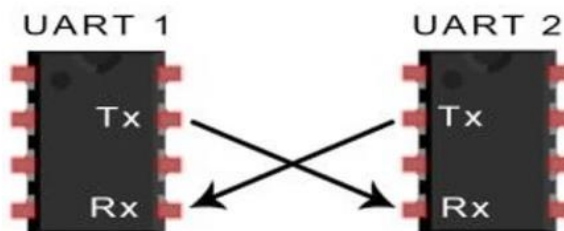


Figure 3.2 UART Communication Diagram

UART supports bi-directional, asynchronous, and serial data transmission. UART only allows communication one way at a time and does not allow for multiple transmitting and receiving systems. UART also does not allow more than 8 bits per message being sent and is known for low data transmission speeds.[7] SPI communication has uses four wires/ports to connect devices which are the MOSI/SDI (master-out-slave-in/serial-data-in) pin, MISO/SDO (master-in-slave-out/serial-data-out) pin, SCLK (serial-clock) pin, and SS/CS (slave-select/chip-select). SPI communication allows for multiple devices to be connected to the master. The SPI communication has simple and inexpensive hardware requiring two shift register which are simple logic circuits. SPI communication between devices needs to be well established before integration because it does not allow devices to communicate at the same time. Therefore, the chip select pin is needed for all devices to establish which device communicates at a specific time. This increases the communication speed and increase in number of pins used because of the dedication to one slave device but can get problematic with multiple devices and switching between peripherals [5]. Figure 3.3 [10] shows the basic SPI communication interface.

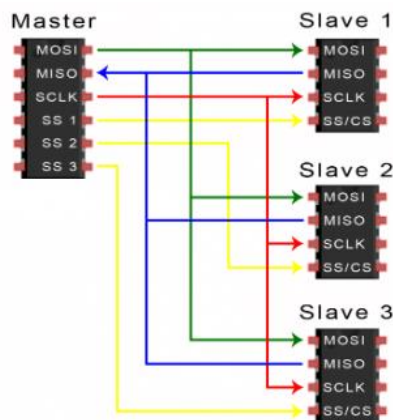


Figure 3.3 SPI communication diagram

I2C is a bidirectional two-wire synchronous serial bus and requires only two wires Serial Clock Line and Serial Data line to transmit information between devices on a bus. I2C uses an address system which assign a unique address of 7-10 bits to the devices connected to allow data transmission at the same time. Because of this architecture, I2C is slower than SPI but allows for over 100 devices to be connected and communicate simultaneously with just the two pins. The serial clock line synchronizes the data being sent to which decides when the master or slave device is transmitting or receiving data. This is done by the master sending a read or write command to the unique address of the intended device which switches the SCL and SDA lines high or low [8]. Basic I2C communication can be seen in Figure 3.4 [11] below.

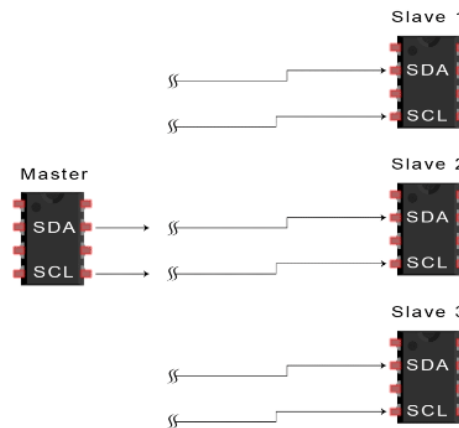


Figure 3.4 I2C communication Diagram

Our IoT device has the 5 different sensors which all can communicate via I2C. I2C has the advantage of requiring less pins, allows for all our devices to communicate via the two pins SCL, SDA, and allows for flexibility in the future if we add more devices. The slower communication speed will work fine with our implementation

because the data transfer will be quick enough to allow the consumer to react to any severe air quality. For our IoT system design we have decided to use the I2C protocol.

Tables with the pin descriptions for each breakout board used for prototyping and testing the overall circuit were made to determine which wires would be connected to the corresponding pin. Below are the pinout diagrams for each component and breakout board used in our prototype system.

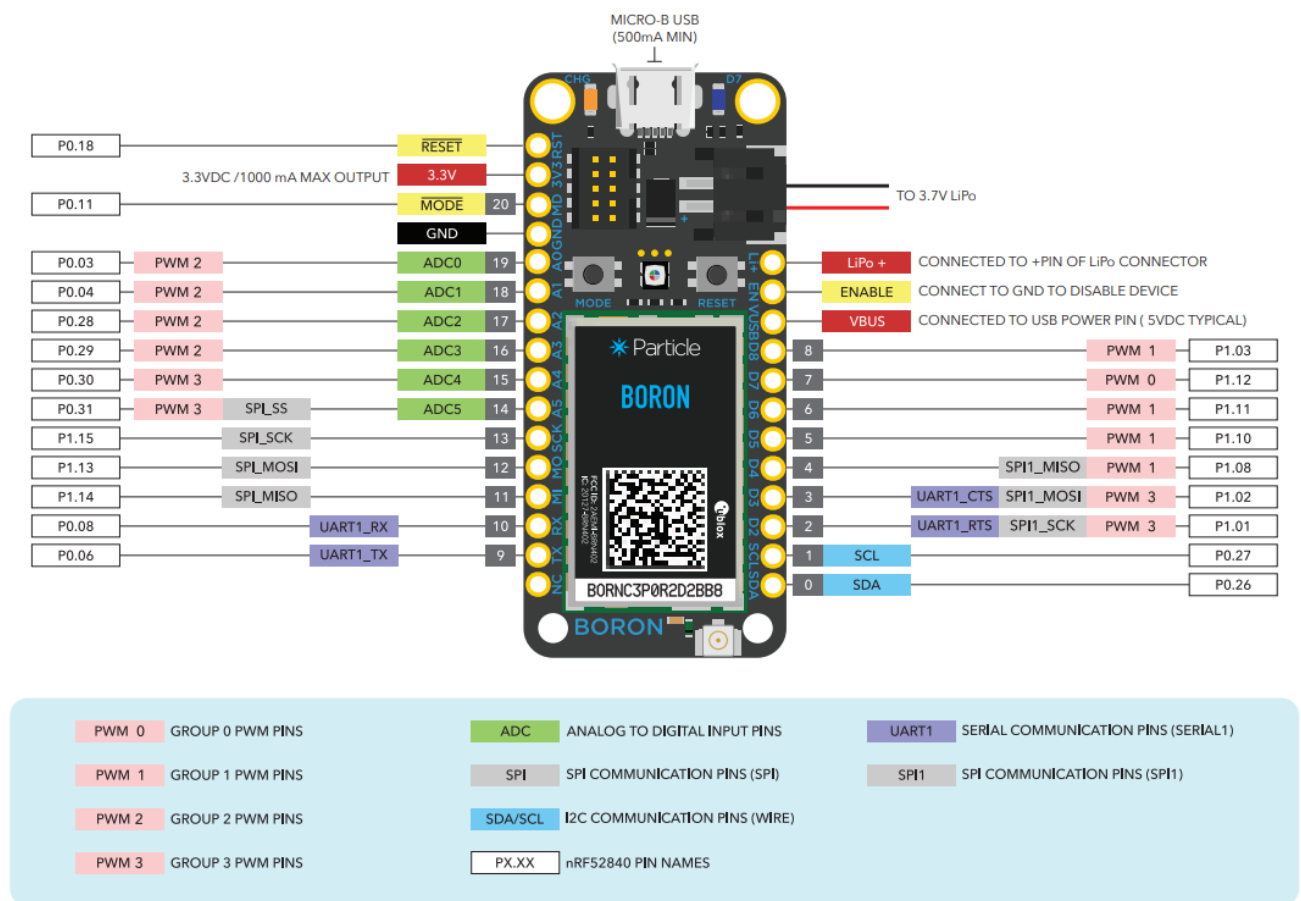


Figure 3.5 Particle Boron Pinout Diagram

Table 3.2 BME280 Breakout Board Pin Out Diagram

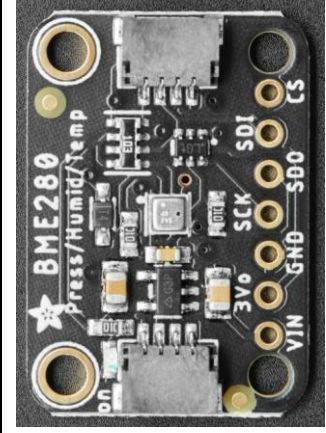
	VIN	3.3V Input
	3Vo	3.3V Output
	GND	Ground
	SCK	SPI/I2C clock
	SDO	Output data for SPI
	SDI	SPI/I2C data
	CS	Chip Select for SPI

Table 3.3 SGP30 Breakout Board Pinout Diagram

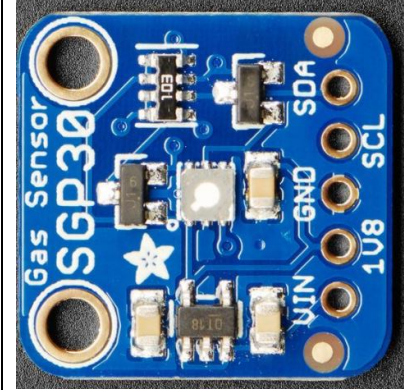
	VIN	3.3V Input
	1V8	1.8V Output
	GND	Ground
	SCL	SPI/I2C clock
	SDA	SPI/I2C data

Table 3.4 SPARKFUN GPS Breakout Board Pinout Diagram

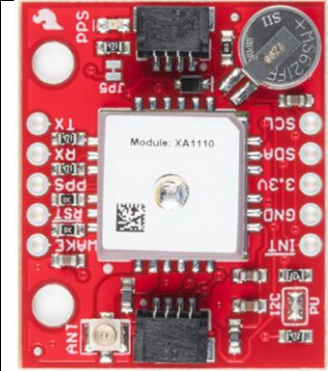
	3.3V	3.3V Input
	GND	Ground
	SCL	SPI/I2C clock
	SDA	SPI/I2C data

Table 3.5 SCD30 Pin Out Diagram

Pin	Comments
VDD	Supply Voltage
GND	Ground
TX/ SCL	Modbus: Transmission line (Push/Pull with 3V level) I ² C: Serial clock (internal 45kΩ pull-up resistor, pulled to 3V)
RX/ SDA	Modbus: Receive line (Input must not exceed 5.5V) I ² C: Serial data (internal 45kΩ pull-up resistor, pulled to 3V)
RDY	Data ready pin. High when data is ready for read-out
PWM	PWM output of CO ₂ concentration measurement (PWM not supported yet)
SEL	Interface select pin. Pull to VDD (do not exceed 4V, use voltage divider in case your VDD is >4V) for selecting Modbus, leave floating or connect to GND for selecting I ² C.

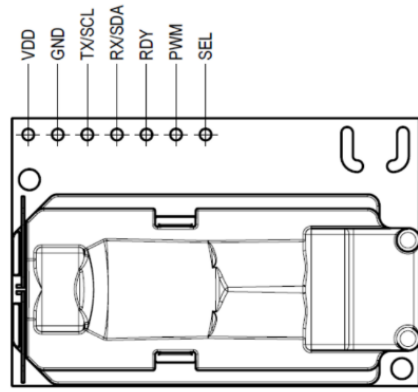
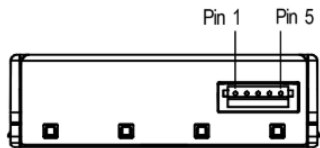


Table 3.6 SPS30 Pin Out Diagram

Pin	Name	Description	Comments
1	VDD	Supply voltage	5V ± 10%
2	RX	UART: Receiving pin for communication	TTL 5V and LVTTTL 3.3V compatible
	SDA	I ² C: Serial data input / output	
3	TX	UART: Transmitting pin for communication	TTL 5V and LVTTTL 3.3V compatible
	SCL	I ² C: Serial clock input	
4	SEL	Interface select	Leave floating to select UART Pull to GND to select I ² C
5	GND	Ground	



3.2 Initial Prototype Circuit

When all the breakout boards were acquired, a breadboard-based circuit was built using a standard breadboard and jumper wires. Some of the design requirements that had to be experimented were the pull up resistors on the SCL and SDA lines for each sensor/breakout board. The pullup resistors pull the specific SCL and SDA lines high when it is not driven low by the open-drain interface. The value of the pullup resistor is an important design consideration for I²C systems as an incorrect value can lead to the

sensor/breakout board to not communicate with the microcontroller. The pull up resistor prevents the I2C pins of the microcontroller to be driven low. The level which the valid logical low (V_{OL}) can be read by input buffers of an IC determine the minimum pullup resistance required for proper communication. This relationship can be shown in equation 1 from the pull up resistor datasheet.

$$R_P(\min) = \frac{(V_{CC} - V_{OL}(\max))}{I_{OL}} \quad (1)$$

The maximum pullup resistance is limited by the bus capacitance (C_p) due to the specific rise time of the IC within the desired system. If the pullup resistor is too high, then the logical high level may not be reached to allow proper data transfer. The maximum rise time of the IC is needed to determine the maximum pull up resistance. The response of an RC circuit on the bus line can be used to determine the rise to be used for the pull up resistance value. The following equations 2-6 show how to calculate the pull up resistance value from the pull up resistor datasheet.

$$V(t) = V_{CC} \times \left(1 - e^{-\frac{t}{RC}} \right) \quad (2)$$

For $V_{IH} = 0.7 \times V_{CC}$:

$$V_{IH} = 0.7 \times V_{CC} = V_{CC} \times \left(1 - e^{-\frac{t_1}{R_P \times C_b}} \right) \quad (3)$$

For $V_{IL} = 0.3 \times V_{CC}$:

$$V_{IL} = 0.3 \times V_{CC} = V_{CC} \times \left(1 - e^{-\frac{t_2}{R_P \times C_b}} \right) \quad (4)$$

$$t_r = t_2 - t_1 = 0.8473 \times R_p \times C_b \quad (5)$$

$$R_p(\text{max}) = \frac{t_r}{(0.8473 \times C_b)} \quad (6)$$

With these equations and information from the technical datasheets from a sensor using I2C protocol, you can determine the correct pull up resistor value. Once the correct pull up resistor is identified, you must place it in parallel of the supply voltage of the intended chip and the SCL and SDA line respectively as visualized in the following Figure 3.6, where V_{in} is the input voltage, R_p is the pull up resistors, SCL is the serial clock and SDA is the serial data. The more devices you add, you may need to add more resistors in parallel or increase the values of the initial pull up resistors.

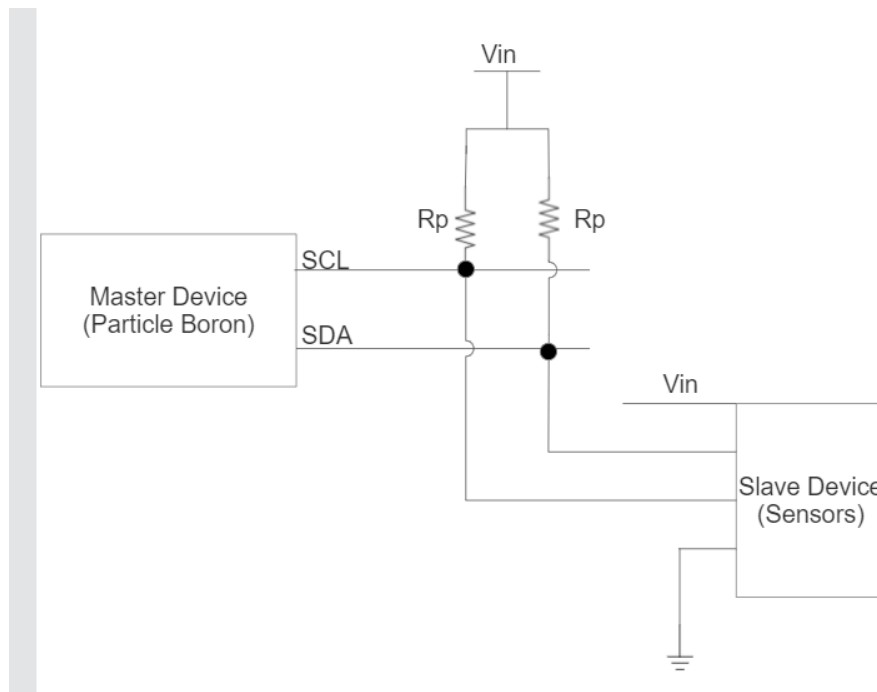


Figure 3.6 Pull up Resistor I2C Bus Topology

For our initial breadboard based prototyped system, most breakout boards had a built-in pull up resistor value to allow for proper communication. For the SPS30 Sensirion sensor, 10Kohm pullup resistors were needed to allow for proper communication. Pull up resistance values would need to be calculated again during our own electrical circuit and printed circuit board design which will be discussed in later sections.

With all the necessary circuitry in place for the communication lines, the power and ground were needed to be connected to all the breakout boards. The 3.3V out pin from the Particle Boron is enough to power on all the devices within our IoT device. The only additions needed for this circuit was a 5V Booster circuit to provide 5V to the SPS30 sensor which requires a 5V input and a logic level shifter to shift the logical levels of the SPS30 SDA and SCL lines back to 3.3V. Every microcontroller has a voltage rating for their I2C communication lines. The Particle Boron logical high-level maxes out at 3.3V volts. Any voltage rating higher than the Particle Boron's specification, then there is serious risk of permanent damage to the I2C lines. Every sensor I2C lines will output a logical level high value with respect to its VIN voltage. In our system, the SPS30 sensor needs the 5V to be powered but the normal logical level high of 5V would not meet the requirements of our microcontroller. Therefore, the logic level shifter was included in the circuit. The wiring diagram for the breadboard-based prototype circuit including all connections between all sensors, GPS, and microcontroller is shown in Figure 3.3.

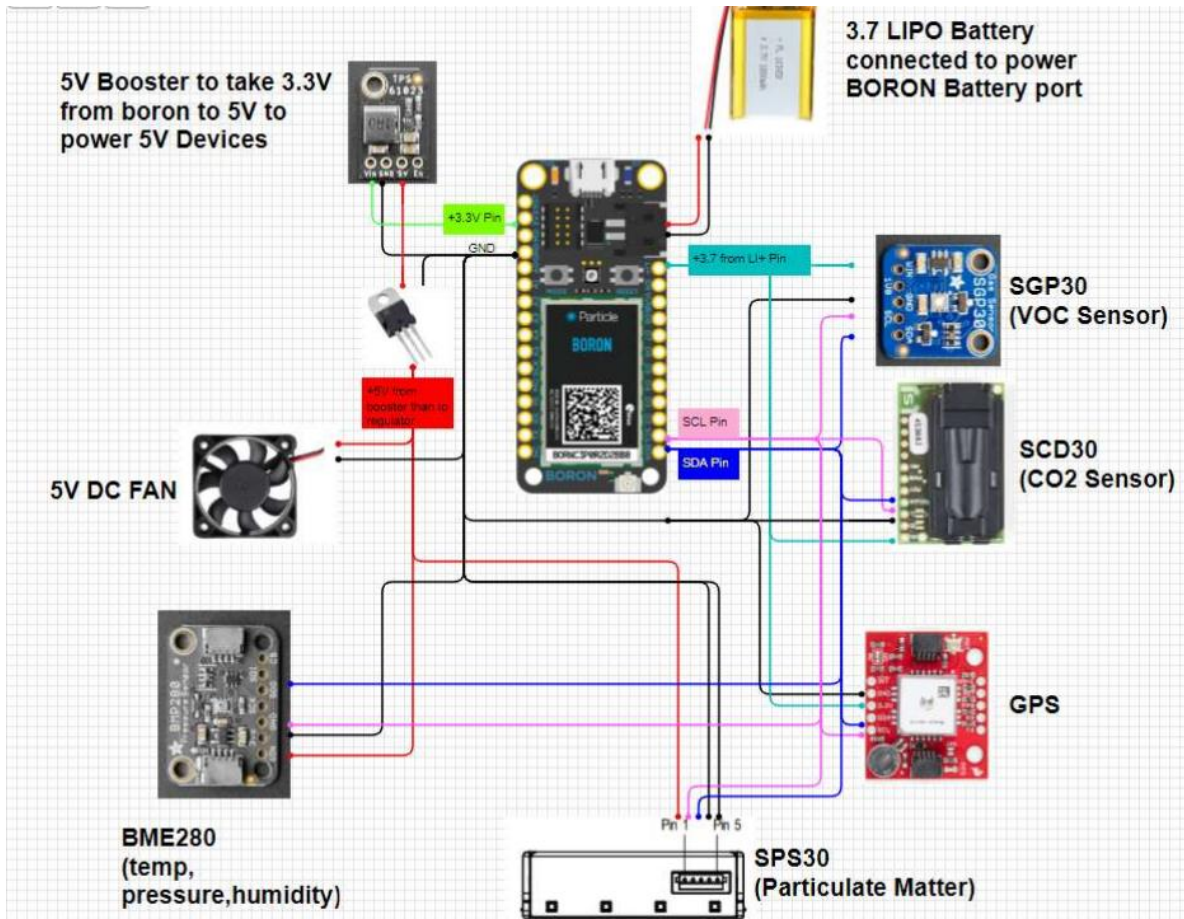


Figure 3.7 Breadboard Prototype Circuit Diagram

After developing this circuit diagram, the system was tested on a breadboard to ensure functionality which will be discussed in our next chapter. After our breadboard-based prototype circuit was tested and proved to be operational, then we created our own proprietary circuit designs including only the sensing ICs of each tested breakout board circuit and our own passive components i.e., capacitors, resistors, transistor, connectors etc. on EAGLECAD software. Following this, the design was then sent to the JLCPCB Company for manufacturing.

CHAPTER FOUR

SOFTWARE INTEGRATION AND TESTING

4.1 Individual Component Testing and Software Development

After designing the breadboard-based circuit, software integration for the entire system needed to be performed. Particle uses an online software called Web IDE that makes editing code, compiling, and flashing devices over the cellular network connection seamless. All the sensors have Arduino C programming language-based libraries, but the Particle IDE uses C++ programming language. This means changes in syntax from existing libraries and the individual/full system codes that include the functions we will be using for each sensor. Using the Particle IDE, we setup and tested functions for each sensor to verify before designing a miniaturized printed circuit board that includes all the sensors and a proprietary circuit design including passive and active components.

4.1.1 Microprocessor Setup

To setup the Boron Particle, we must register it with a particle account. To do this, we needed a usb cable for power, and external antenna and an iOS Bluetooth enabled device. The iOS device finds the Bluetooth connection to the Particle Boron to find a unique data matrix from the microprocessor. Then the cellular connection is used to link to the account used on the iOS device to the Particle Boron. To verify connection, a simple LED blinking code was flashed onto the device to turn the onboard LED on and off. This and all future software will have library inclusions, information definitions and declarations, variable creation, setup function, and repeated loop function calling a specific function/command from the included libraries. The libraries include code with

commands that could be used for sensors and actuators, communication protocol functions, and memory structures. The information definitions and declarations are where we label the variables to an air quality parameter for future use. The variable creation is where we tell the code the name of the air quality parameter we are trying to read and its data size and where we want it to be sent in the system. The setup function and repeated loop is where we take the command from the library to find the air quality parameter and send the value to the variable, we created to then be published to the particle dashboard continuously. All sensors will use the following libraries shown below as they are needed for basic math functions, particle functions, and I2C functions.

```
#include <Wire.h>
#include <math.h>
#include <Particle.h>
```

4.1.3 Temperature, Pressure, Humidity Sensor Setup

The top of this setup uses the library named “Adafruit_BME280.h”, which has the necessary memory structures, communication protocol, and functions to initialize this sensor. This line of code can be shown below:

```
#include <Adafruit_BME280.h>
```

All the remaining sensors will require a similar line of code for their respective libraries which are pre-existing and created by our team. We then initialize the library by calling a specific name within the library called bme as shown in the code below.

```
Adafruit_BME280 bme; //Select I2C for the BME280 (temp, pressure, humidity)
```

Now when we call commands from the library, we can just write bme.command, where command can be the function we intend to use from the library. A connection bit was

created to determine if the sensor is connected and found by the microcontroller which is shown below:

```
int BME_connected = 0; // the BME280 connection bit

// Connecting to the Temp, pressure, humidity sensor
if (!bme.begin(0x77))
  Particle.publish("BME280", "Could not find a valid BME280!");
else
  BME_connected = 1;
```

Every sensor will have this connection verification in their setup. Before creating a loop, the variables used, and sizes of variables need to be stated in the code. For the BME THP sensor, we declare the Humidity, Temperature, and Pressure and allow for 64 bits of data for each, which is known as a double value. As stated previously, we will be using I2C communication, to initialize this in the code we must use `Wire.Begin()`, determine a communication rate with `Serial.Begin()`, then find the communication address. The variable size, communication address, and communication speed can all be found in the sensor documentation. In the setup function, the I2C interface address was initiated at address 0x77. Once the setup has been created for our function, we need to write the command to read the temperature, pressure, and humidity from our sensors. This line of code looks like the following lines below:

```
getBme280();

void getBme280() {
  if(BME_connected) {
    data[temp_C] = toC( (110.0/125.0) * toF(bme.readTemperature()) );
    data[pressure] = (double)bme.readPressure()/100;
    data[humidity] = bme.readHumidity()+10;
  }
}
```

These lines of code are the essential lines to allow the sensor to communicate the correct variable to the microcontroller. These are all combined with the necessary C++ programming syntax of IF ELSE statements, function calling, and variable initialization. Then using the Particle. Publish command, the data[temp_C], data[pressure], and data[humidity] can be published to the particle console dashboard. This loop is then constantly updated with new readings and sent to the particle dashboard. The following sensors have a similar structure for their software setup.

4.1.4 Carbon Dioxide Sensor Setup

The Carbon Dioxide sensor, the SCD30 has an existing library called “SparkFun_SCD30_Arduino_Library.h”. A vector of size one was declared because we want the one carbon dioxide reading from the sensor as a double value. The function from the library was SCD so the required call set up was as shown in the line of code below:

```
SCD30 SCD; // Create the SCD30 module
```

After creating the module and beginning I2C communication with sensor and the microcontroller, then we call the required command to get the CO2 values consistently in a loop which are shown below:

```
void getScd30() {  
  if(SCD_connected) {  
    if(SCD.dataAvailable()) {  
      data[temp_F] = (110.0/125.0) * toF(SCD.getTemperature());  
      data[CO2] = (1.125) * (double)SCD.getCO2();  
    }  
  }  
}
```

In the loop function the temperature the concentration of carbon dioxide (given in ppm) is constantly read and sent to the particle publish command in the code.

4.1.5 Particulate Matter Sensor Setup

Although the hardware setup for the particulate matter sensor, the SPS30 is different than the other sensors, the software set up will remain like the previous sensors. The library used for this software setup is named “SPS30.h” and the name of the module for I2C communication is SPS and this line of code looks like the following lines of code:

```
SPS30 SPS;      // Create the SPS30 module
```

The SPS30 sensor needs 4 variable initialized for the different ranges that the sensor can sense. These variable are cPM1, cPM2_5, cPM4, and cPM10 and are set as float values which allow for 32 bits of data awaiting to be communicated. Then the loop is created using the command getMass and setting the arrays to their respective range values. These lines of code for the loop are as shown below:

```
void getSps30() {  
  if( SPS.dataAvailable() ) {  
    SPS.getMass(mass_concen);  
    data[cPM1] = mass_concen[0];  
    data[cPM2_5] = mass_concen[1];  
    data[cPM4] = mass_concen[2];  
    data[cPM10] = mass_concen[3];  
  }  
}
```

This loop continuously finds the particulate matter data and assigns it to the variables initialized to be published on the particle’s dashboard.

4.1.6 Volatile Organic Compound (VOC) Sensor Setup

The Volatile Organic Compound sensor, SGP30 sensor uses a library called “Adafruit_SGP30.h”. The variable initialized for this function is labeled as VOC and the module name created is sgp as shown below:

```
Adafruit_SGP30 sgp; // create SGP30 module
```

The command used to read the data to send to the sensor is called `sgp.TVOC` which is from the library. The function used for the VOC sensing is shown in the line of code below, which retrieves the VOC as a double value:

```
void getSgp() {  
  if (sgp.IAQmeasure()) {  
    data[VOC] = sgp.TVOC;  
  }  
}
```

This is the last air quality parameter sensor to be setup before the integration for the entire system to be streamlined and output every variable from each sensor.

4.1.7 GPS Setup

The Global Positioning System, GPS sensor has the greatest number of commands and options out of all the components on the board. The accuracy of the GPS sensor is important for integration to allow for accurate location documentation for poor to severe air quality detected from the system. The GPS sensor uses two libraries called “SparkFun_I2C_GPS.h” and “TinyGPS.h”. The module created was labeled `myI2C` and `gps` as shown in the code below:

```
#include "SparkFun_I2C_GPS.h"  
#include "TinyGPS.h"
```

The two variable we are initializing are the latitude and longitude labeled `lat` and `lng` as double values in the code. The main function with the appropriate commands is shown below:

```

int gpsUpdate() {
  if(GPS_connected) { //if the GPS is connected...
    // read the GPS and encode it into the parser
    while( myI2CGPS.available() ) { gps.encode(myI2CGPS.read()); }
    if( gps.time.isUpdated() ) { //If the GPS has sent new data
      if( gps.location.isValid() ) { // and the data is valid
        lat = (double) gps.location.lat(); // get the latitude
        lng = (double) gps.location.lng(); // get the longitude
        char g[50]; sprintf(g,"%2.5f,%2.5f",lat,lng); // create the data packet
        Particle.publish("GPS",g,PRIVATE,WITH_ACK); // send the data packet
        GPS_count=0;
        return 0; //return success
      }
    }
    else { // if GPS has sent data but it's not valid
      GPS_count++;
      Particle.publish("!Valid","GPS location not yet valid");
      if(GPS_count>4) {
        GPS_count=0;
        if(lat!=0 && lng!=0) {
          value = (double)((range)?0.00001:0.0);
          char g[50]; sprintf(g,"%2.5f,%2.5f",lat+value,lng); // create the data packet
          Particle.publish("GPS",g,PRIVATE,WITH_ACK); // send the data packet
          range=!range;
        }
      }
    }
    return 0;
  }
}
return 1; // if the GPS is not connected or the data has not updated then return failure
}

```

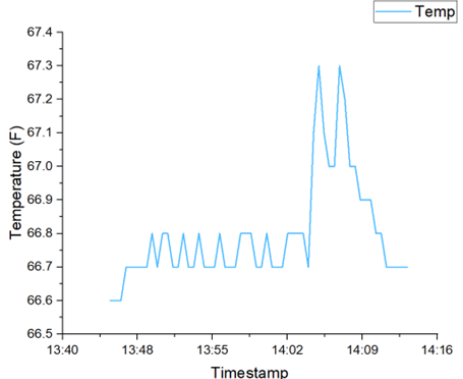
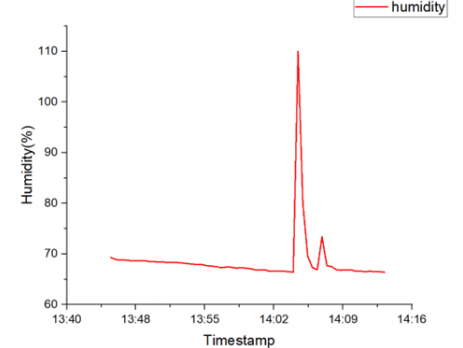
This function reads the latitude and longitude as double values and prepares the data to be sent to the particle dashboard. An IF statement is also included to verify that the GPS is communicating properly and publishes a statement to the dashboard if it is not. Also included are functions to get the time in whichever time zone the system is located which will be in the index of this paper along with all the lines of code not shown for each sensor.

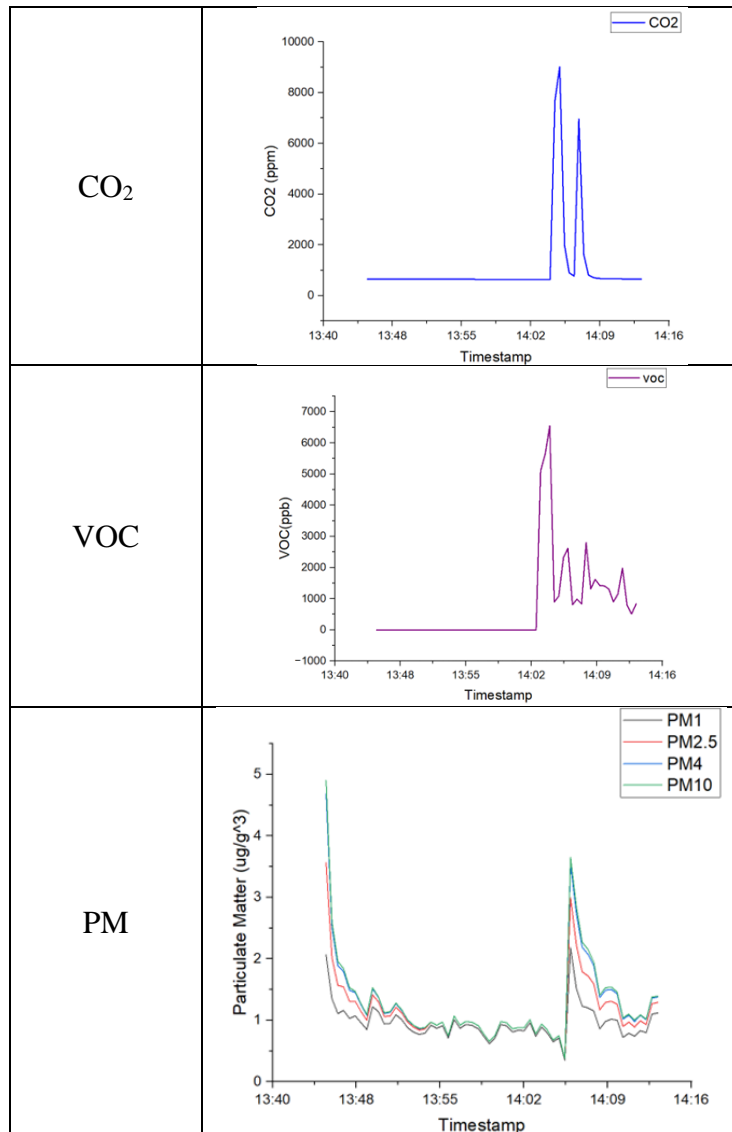
4.2 Combined Sensor Component Testing

Once all sensors have been setup by the particle Web IDE individually, a singular full system code was created to streamline the data with one full code. This code included the library inclusions, variable declaration and creation, functions, and loops all in one

code which can be seen in the Appendix G of this paper. With this full system code, we were able to begin testing the different sensors sensitivity, accuracy, and response time. This test included impulse values for the sensors based the response of each sensor. These impulses come in the form of concentrated dust from a common household powder, smoke particles from a lighter, and alcohol and cleaning products commonly found in one’s home. The resulting graphs are shown in Table 4.1. This test showed that the Particle Boron can communicate with each sensor and obtain all the required data we have made a requirement for our system timely and efficiently.

Table 4.1 First System Test Resultant Graphs with Steady State and Impulse Inputs

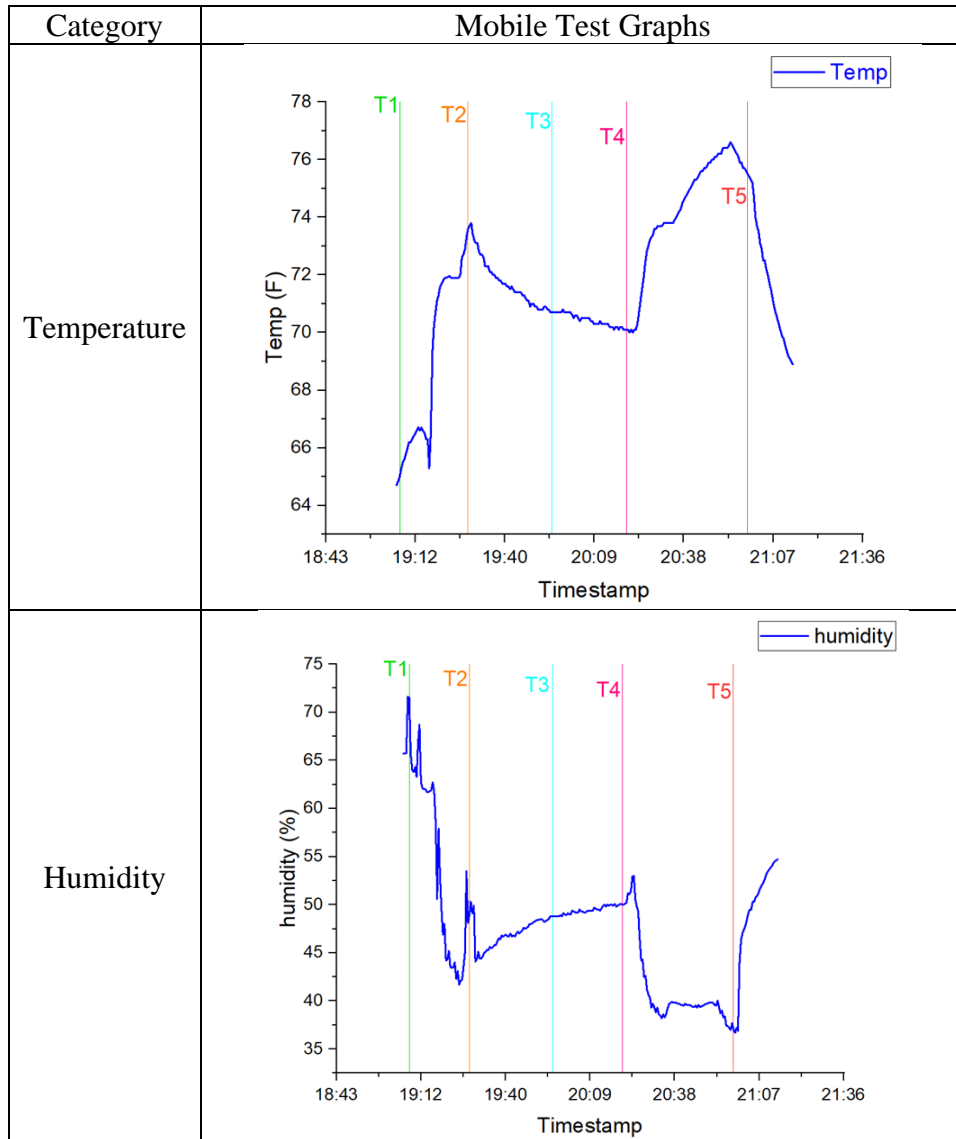
Category	Impulse Inputs
Temperature	 <p>The graph displays Temperature (F) on the y-axis (ranging from 66.5 to 67.4) against Timestamp on the x-axis (ranging from 13:40 to 14:16). A blue line labeled 'Temp' shows a steady state around 66.75 F with small oscillations until approximately 14:02. At this point, there is a sharp impulse spike reaching a peak of about 67.3 F, followed by a gradual return to the steady state.</p>
Humidity	 <p>The graph displays Humidity (%) on the y-axis (ranging from 60 to 110) against Timestamp on the x-axis (ranging from 13:40 to 14:16). A red line labeled 'humidity' shows a steady state around 68% until approximately 14:02. At this point, there is a sharp impulse spike reaching a peak of about 110%, followed by a gradual return to the steady state.</p>

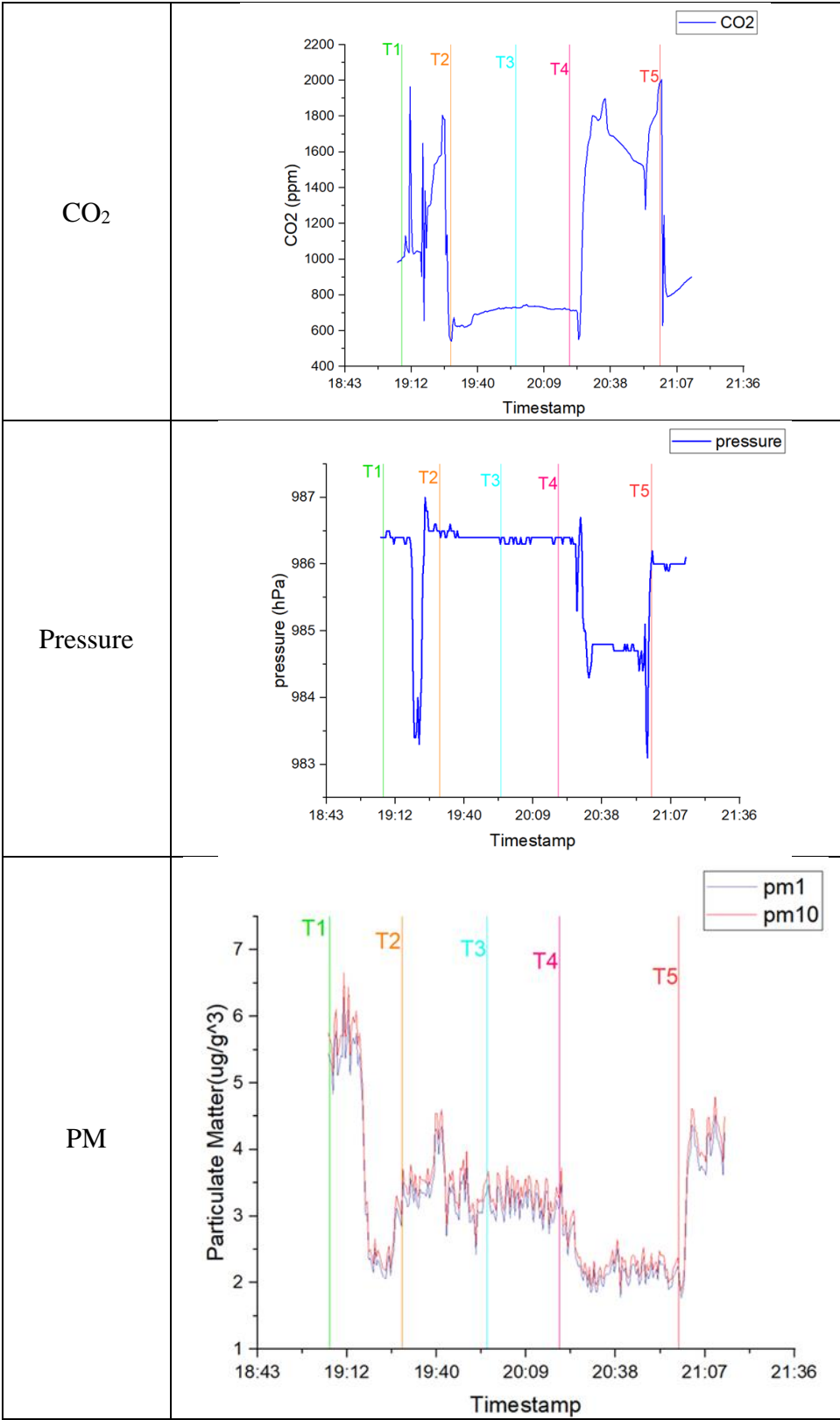


A test was also done to test the GPS and all sensors to simulate a common scenario like going out for dinner. This test allows us to analyze the system performance in different locations and scenarios. This test lasted nearly two hours from 19:08 to 20:56. During this test, the system was placed taken from the base location into a vehicle at T1, 19:08. At T2 19:29, the system was taking into the restaurant at to table party was seated at, T3 19:56, the food arrived to the table, T4 20:20 the system left the restaurant, and T5

20:56 the system arrived at its base location. This test was performed to show how the system can be useful in understanding one's surroundings. The obtained data, with the above-mentioned Times of interest, is shown in the plots in Table 4.2. The maps of the GPS data gathered during this test and a short drive on roads are shown in Table 4.3.

Table 4.2 Mobile Test Data Graphs





CHAPTER FIVE

SENSOR DATA DASHBOARD DEVELOPMENT

5.1 Data Stream Creation and Handling

After developing the software for the system, software able to gather and collect the data during all hours of operation needed to be made. The Particle dashboard displays the data in real time via the cellular network connection, but it does not store the data for data analysis. Losant is a software that allows a particle microcontroller to send the events being sent to the particle dashboard to the Losant data visualizer. The communication code from each individual component's code in the particle web IDE was used to create a memory and data transfer system using Losant. This was determined to be a ten-element array with data type {double} to allow for the storage of large decimal values. The uses of each element in this array are defined at the start of the software as shown below:

```
#define temp_F 0
#define temp_C 1
#define pressure 2
#define humidity 3
#define CO2 4
#define VOC 5
#define cPM1 6
#define cPM2_5 7
#define cPM4 8
#define cPM10 9
```

This ten-element data array was then used in the creation of a formatted JSON string to allow for the online console to parse this string into its multiple component

values to add to the running graphs of the data. This JSON string creation is shown in the code below:

```
String out = String::format ("{\"temp\":%.1f,\"HMD\":%.1f,\"press\":%.1f,\"CO2\":%.0f,\"VOC\":%.0f, \"cPM1\":%.2f,\"cPM2_5\":%.2f,\"cPM4\":%.2f,\"cPM10\":%.2f}", data[temp_F], data[humidity], data[pressure], data[CO2], data[VOC], data[cPM1], data[cPM2_5], data[cPM4], data[cPM10]);
```

This data string is 80 bytes of data each time it is transferred from the Boron to Losant.

The data that is received by Losant follows the order of the string creation, which is temperature in Fahrenheit to the tenth decimal place, percentage of humidity to the tenth decimal place, pressure in hectopascal as a whole number, concentration of Carbon Dioxide (CO₂) as a whole number in parts per million, concentration of Volatile Organic Compounds (VOCs) as a whole number in parts per billion, the concentration of particulate matter in the air with radii of 1µm, 2.5µm, 4µm, and 10µm (in µg/m³) to two decimal places. This string with the air quality parameters and a separate string with GPS location is created by the Particle Boron and sent to the Losant IoT software.

5.2 Losant and Data Processing

Losant takes the data from the intended micro-controller and visualizes the data with easy-to-understand graphs for analysis. This is done by the creation of webhooks that are messages sent to and from services using the internet and URLs. In our case we created webhooks in the Particle Integrations tab and in the Losant website to create the link between the two. These webhooks take the BLOB of string data we created and stores it in the specific value we declare it as in the Losant software. Once the webhooks for communication were created, then a workflow needs to be declared in the Losant

software to be able to process and visualize the information. The workflow for our device is shown below in figure 5.1.

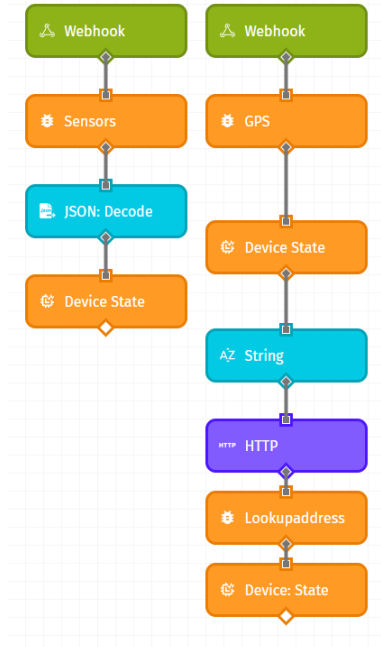


Figure 5.1 Losant Data Workflow

Two workflows were created, one for our original string creation and one for our GPS location. The webhook layer finds the webhook URL, the orange debug layer is where you declare how the information from the webhook is named in the Losant environment. The Blue layer takes the JSON string format and creates its own arrays that can be customized and declared as the air quality variables we are sensing. Then the device state layer is where we take the arrays and match them with the JSON string, we created for the data transfer. An additional layer was created for the GPS location to link the location to an address using google maps URL which is a feature Losant has built in. The following Figure 5.2 shows an example of how the attribute made from Losant was matched with the JSON string label.

Attribute	Value
pm1	{{stuff.cPM1}}
pm10	{{stuff.cPM10}}
pm25	{{stuff.cPM2_5}}

Figure 5.2 Device State Layer

Once this workflow was established in Losant, the data was able to be received and visualized from our Particle Boron. Figure 5.3 shows the block diagram of the path our data takes from our system to an online service that allows one to access the information in real time from anywhere in the world.

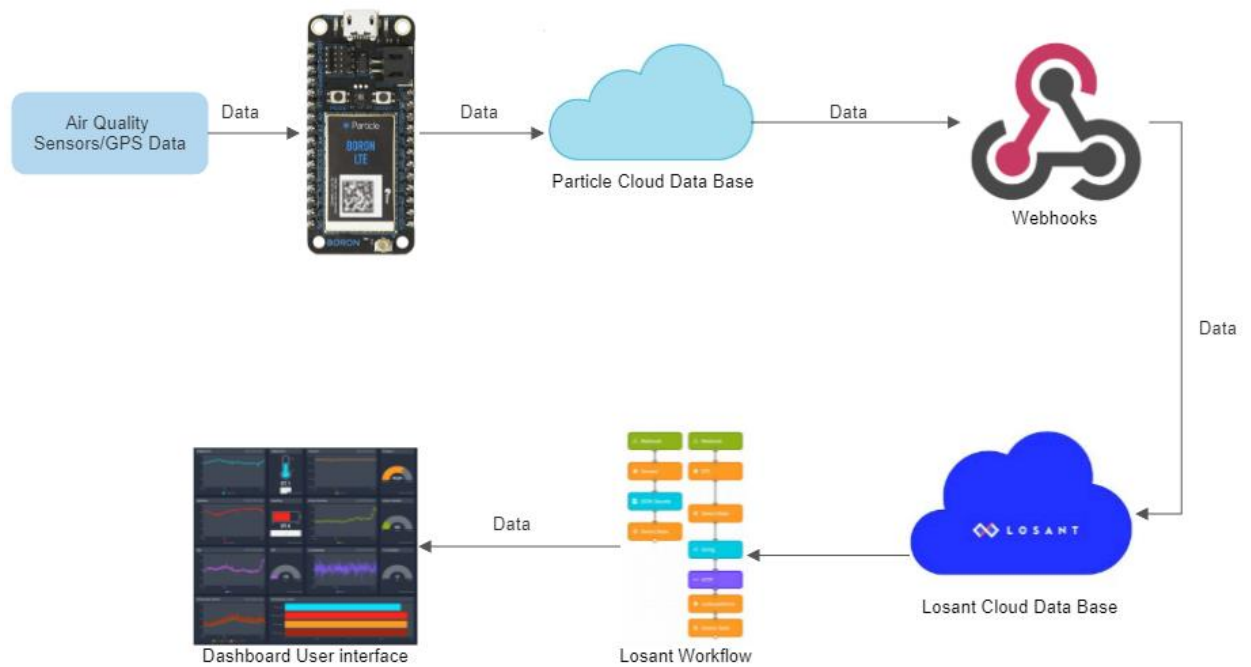


Figure 5.3 Data Path Block Diagram

5.3 Data Visualization and User Interface

The Losant Dashboard allows for real time data visualization. The dashboard allows for customization to allow graphs to display data from 10 minutes to up to 90 days. The following Figure 5.4 shows the dashboard graphs for one hour of operation in average household environment.



Figure 5.4 Losant Data Visualization Dashboard

CHAPTER SIX

FINAL CIRCUIT AND PCB DESIGN

6.1 Sensor Schematic Designs

After completing the software integration and IoT user interface of the entire system. The miniaturization of the system could be performed. To make this system optimal in size, the amount of breakout boards in the system needed to be reduced. To do this, a Printed Circuit Board with the individual sensors and supporting active and passive components needed to be designed. EAGLE, a scriptable electronic design application with schematic capture, printed circuit board layout, and computer aided manufacturing features was used to design the printed circuit board. The first step in designing a printed circuit board in EAGLE, is an electrical schematic of the system needs to be created. To do this, all components used in the system need to have a corresponding footprint within EAGLE. This footprint defines the physical size, physical pin configuration, and pad sizes needed for each component. For example, the Particle Boron will have a design block created for the electrical schematic editor and corresponding footprint as shown in Figure 6.1.

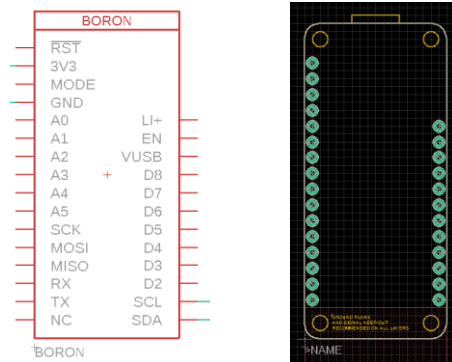


Figure 6.1 Boron Schematic Symbol with corresponding PCB Footprint

EAGLE has some commonly used active and passive components symbol/footprints included with the software but for components that do not exist in EAGLE, the symbol/footprint needs to be created manually. This had to be performed for the Particle Boron, SCD30, SPS30, and BME280. The symbol and footprint for the BME280 is shown in the following Figure 6.2.

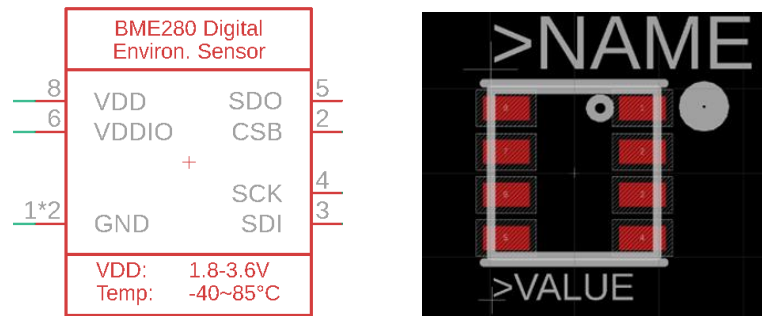


Figure 6.2 BME280 Schematic Symbol with corresponding PCB Footprint

All sensors and components on the board will have this symbol and footprint. After all the symbols and components are made in eagle, then we were able to design the circuit for our system.

Every chip used has an associated data sheet providing recommended circuitry. Using this datasheet, common practices, and the circuit information from the breakout boards used in our prototypes we created the following circuits. The BME280 which is made by Bosch has a recommended I2C circuit configuration as shown in Figure 6.3.

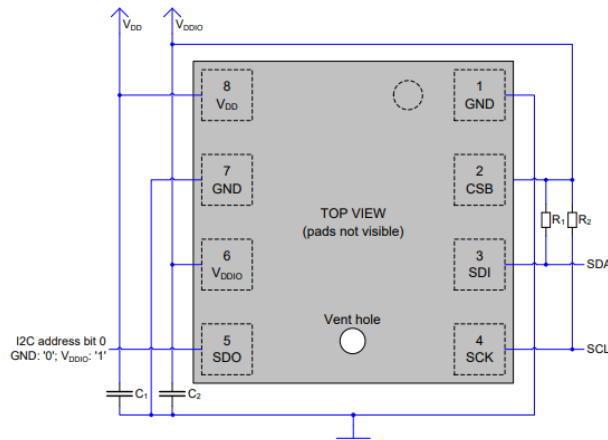


Figure 6.3 Recommended I2C connection circuit for BME280

The R1 and R2 in the diagram are the pull up resistors for the circuit that have previously been discussed and are both 10Kohms. The C1 and C2 values are recommended as 100nF. Using this information, the following circuit diagram for the BME280 was designed as shown in Figure 6.4.

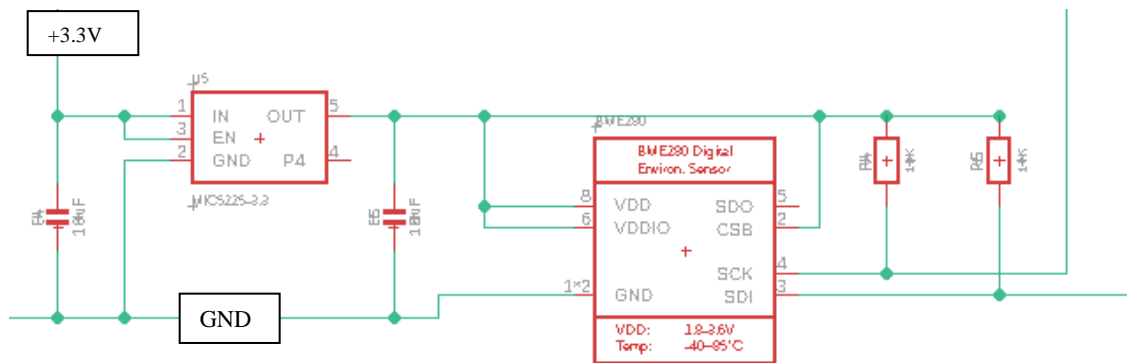


Figure 6.4 Electrical Circuit Schematic for BME280

A low drop out (LDO) regulator was added to the suggested circuit so that we can obtain a lower and ideally stable output voltage from the main power supply of the Particle boron. This was added to add greater stability and allow the BME280 to operate at intended conditions as suggested per the data sheet.

The SGP30 has the following recommended circuitry from the manufacturer, Sensirion as shown in Figure 6.5.

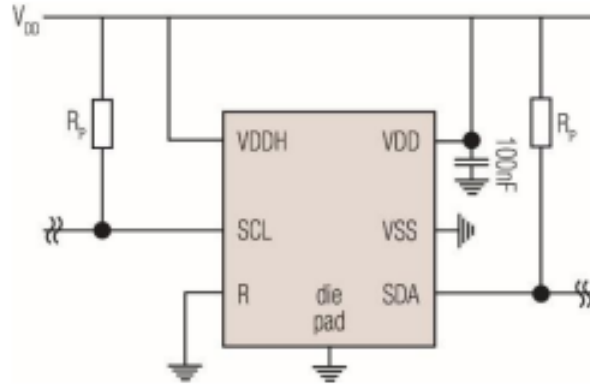


Figure 6.5 Recommended Configuration for SGP30

Given this recommendation and knowledge of pull up resistors, the following circuit shown in Figure 6.6 was created for the SGP30.

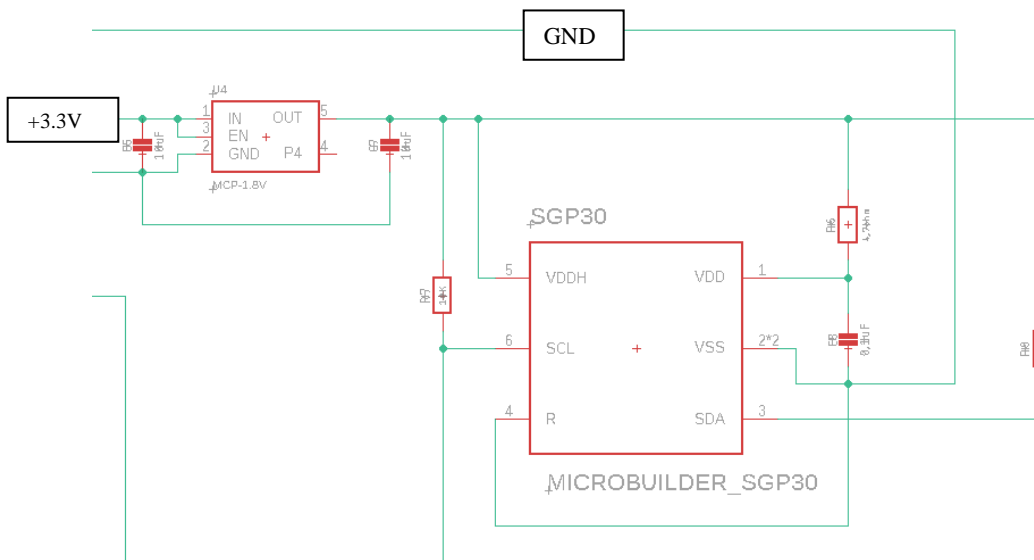


Figure 6.6 Electrical Circuit Schematic for SGP30

This circuit also includes LDO voltage regulator to add stability to the SGP30 sensor, which has 2 capacitors and resistors added for intended operation. The capacitor values are both 10nF and the resistor values is 4.7kohms. The pull up resistance values are both 10Kohms, capacitor value is 01.uF.

The SCD30 does not have individual I2C circuits because the sensing IC is not available to be purchased individually. This means that the SCD30 breakout board will need to be soldered on the printed circuit board. All that is required for the schematic is the electrical symbol and solder pad footprint for the SCD30. The SCD30 Symbol is shown in the following Figure 6.7.

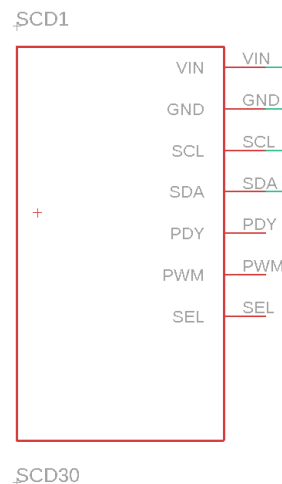


Figure 6.7 SCD30 Electrical Circuit Schematic

The SPS30 like the SCD30 will be soldered onto the printed circuit board but does require some external circuitry to be able to operate properly. The SPS30 requires a 5V input, so a boosting circuit with a voltage boosting integrated chip and its corresponding passive components for operation was designed. This circuit is shown in

Figure 6.8, where the VIN is the 3.3V from the Particle Boron and the VOUT is the 5V needed to power on the SPS30.

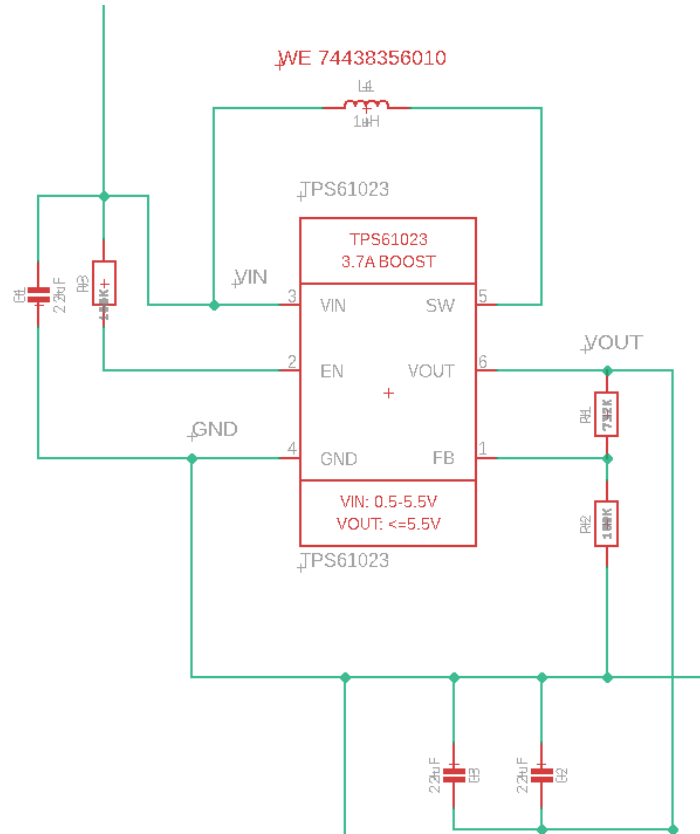


Figure 6.8 5V Boost Circuit Schematic for SPS30 Input

After designing the circuit to power on the SPS30, the output communication signals SCL and SDA needed their corresponding pull up resistors as well as their logic level shifted down to 3.3V per the Particle Boron specifications. To do this, N channel MOSFETs were used to control the communication outputs to 3.3V. This logic level shifting circuit is shown in Figure 6.9.

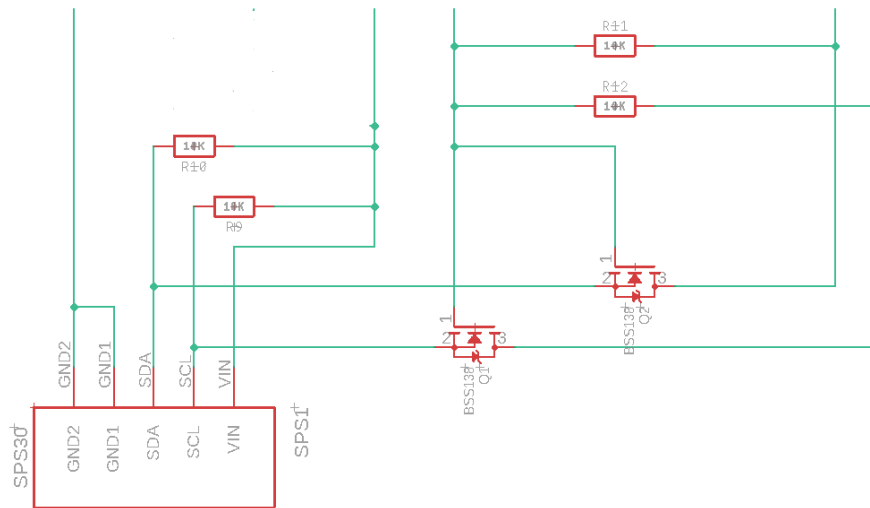


Figure 6.9 Logic Level Shifting Circuit Schematic

The value of the resistors used were 10kohms as suggested from the SPS30 documentation.

Finally, the GPS circuit was designed per the recommendation of the datasheet and the circuitry from the breakout board used by SparkFun as shown in Figure 6.10.

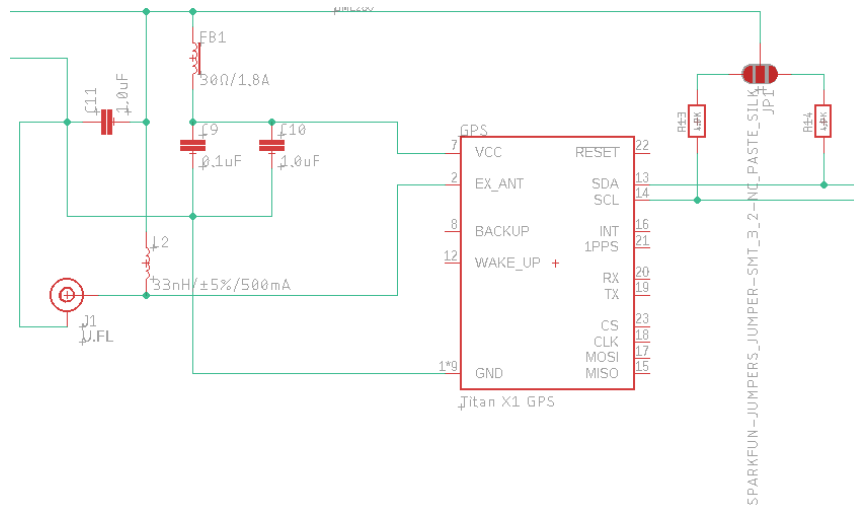


Figure 6.10 GPS Electrical Circuit Schematic

After all the sensor and component circuits were created, we were able to use that and connect them all together to create a full schematic.

6.2 Final Circuit Sensor Schematic Design

After designing all the circuits for each component, they were all connected according to the circuitry of our breadboard-based prototype and values from our electrical circuit schematics. The following full schematic was created so that a printed circuit board could be designed, Figure 6.11. This schematic shows all components inputs and outputs and their connection to the Particle Boron. The 3.3V out of the Particle Boron supplies power to all the components, the SCL and SDA lines connect all communication lines for all the sensors, and the GND provides the grounding.

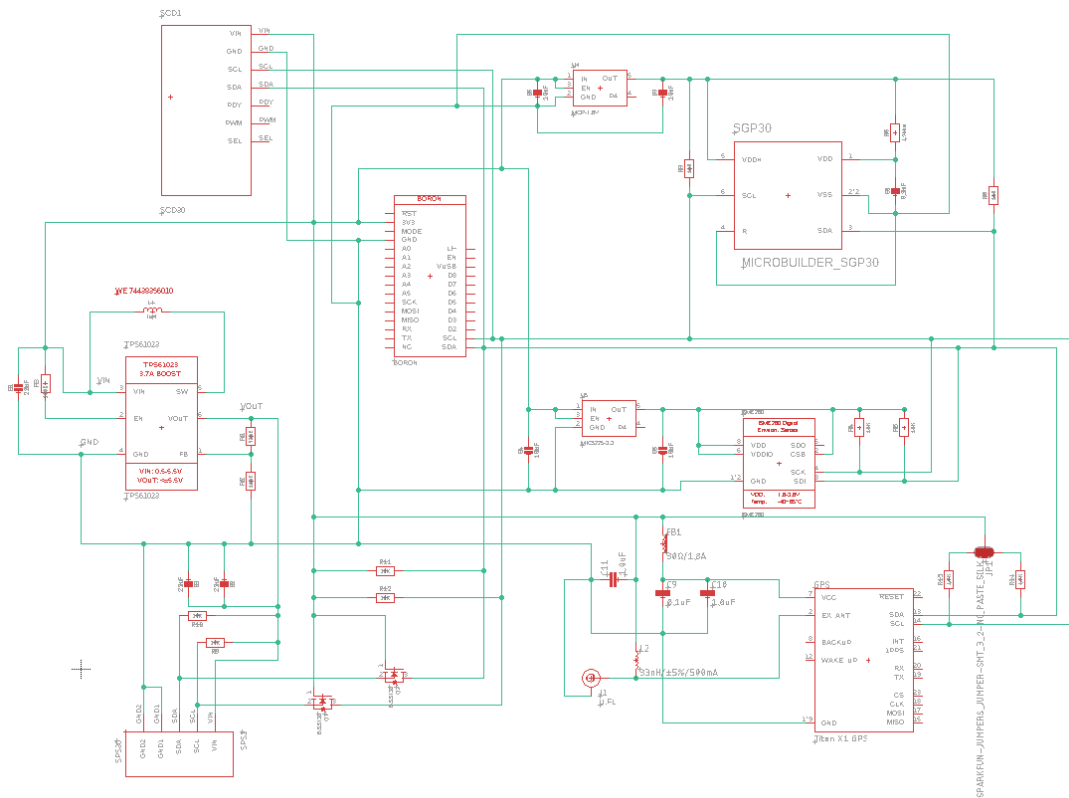


Figure 6.11 Electrical Circuit Schematic of Full System

The Particle Boron is powered by a 3.7V 1800mAh rechargeable Li-Po battery. The Boron has an internal battery charging circuit that will keep the system powered while being disconnected from a usb connection. The systems average and maximum current draw are shown in the table below resulting in an estimated usage time of 8 hours under normal conditions with the 1800mAh battery being used. A full listing of all component power usage is given in Table 6.1.

Table 6.1 Final System Current Draw and Power Consumption

Part	C Norm	C Max
Boron	90	180
BME280	1	5
SCD30	19	75
SPS30	60	80
SGP30	48	70
GPS	27	30
Total Current	~245 mA	~440 mA
Power	~808 mW	~1.453 W

This total current is well below the maximum current output for the Particle Boron 3.3+V output pin. This will allow for future sensor and component additions if desired. This current system with its current battery, sensor, and components has an estimated usage time of 7 hours.

6.3 Printed Circuit Board Design

After creating the Electrical Circuit Schematic for the system, the printed circuit board had to be designed. Using the schematic information and footprint data, EAGLE will transfer the Electrical Schematic into a printed circuit board design environment. This environment takes all the connections and components and lays them out into the

software as shown in Figure 6.12. The yellow wires are just as reference to show how all components are connected.

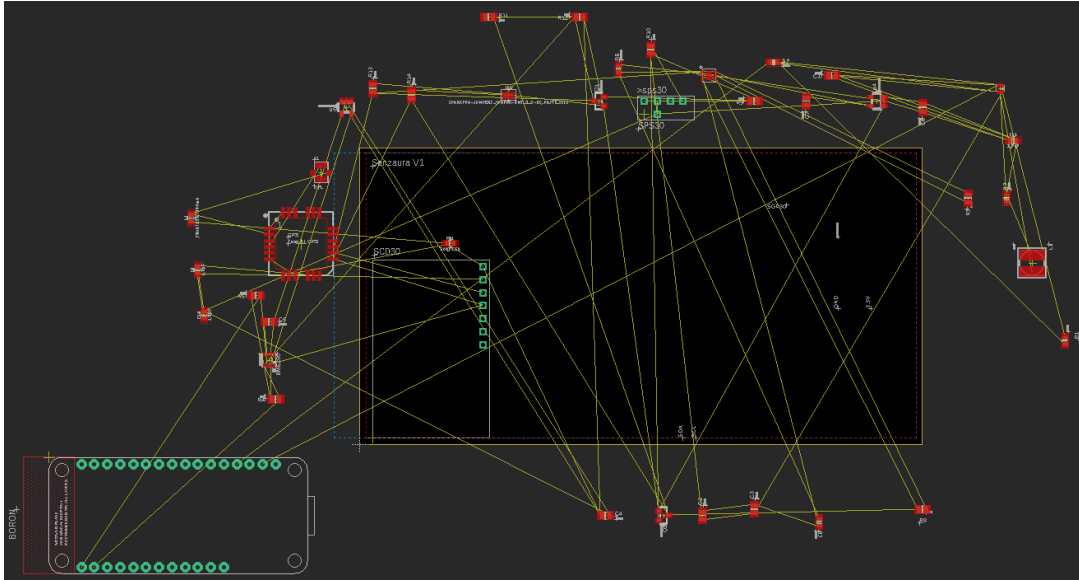


Figure 6.12 All components and connections in PCB software

Once all the components were transferred in, the placement of the components needed to be determined. Each air quality variable sensor and their corresponding components were all placed near each other to reduce the length of the routes and to follow some requirements from the sensor datasheets. The Particle boron was placed on the side of the printed circuit board so that the Li-Po battery could be placed underneath to reduce the size of the full system. The initial placement of the components on the printed circuit board can be seen in Figure 6.13.

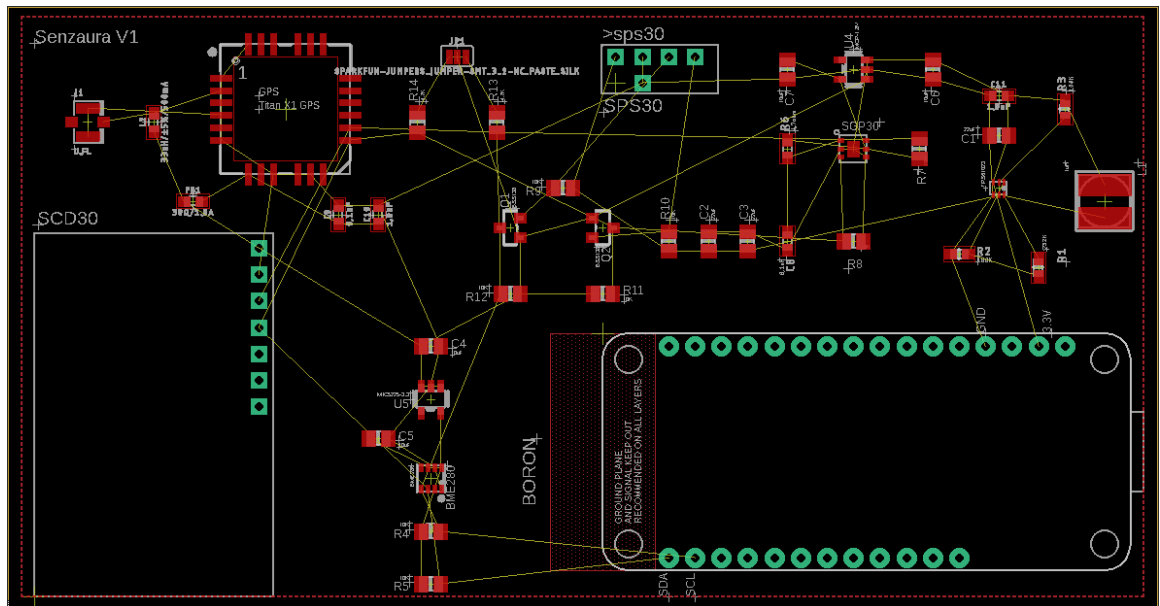


Figure 6.13 PCB Component placement with connections

Once the components are placed on the board, then routing needs to be determined. This routing is how the signals are internally connected by the board. Using the yellow “air” wires that are reference to how components are connected, then you can use a routing feature in the EAGLE software to design these routes. Using common practices and some verification features from the design software, the following routes were designed. The red and blue lines are routes connecting all the components where the blue routes called vias are on the other side of the board to avoid overlapping connections, shorts. This makes the printed circuit board a 2-layer board with a grounding layer in between to ground all the components. After labeling all components and lines, the final printed circuit board design is as shown below in Figure 6.14.

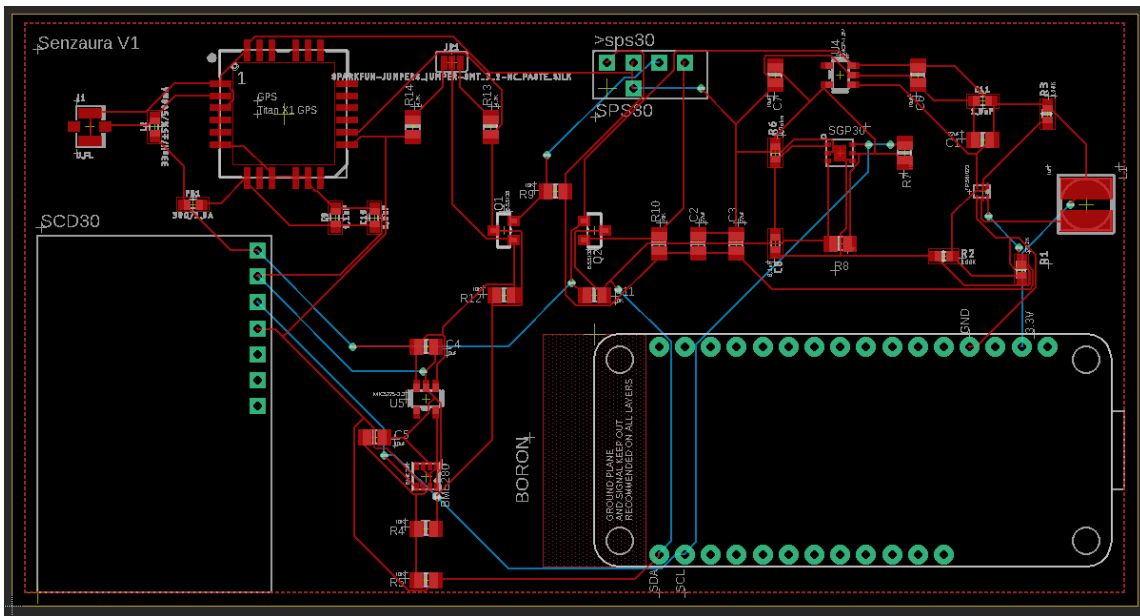


Figure 6.14 Printed Circuit Board Design for Full System

6.4 Printed Circuit Board Fabrication Process

After the design of the printed circuit board, then it was manufactured by PCBWAY. All that was needed was the Gerber files of the PCB design from eagle which can be previewed and visualized as seen in Figure 6.15(top, bottom, grounding layer).

The final PCB size came out to be 4.35 x 2.29 inches.

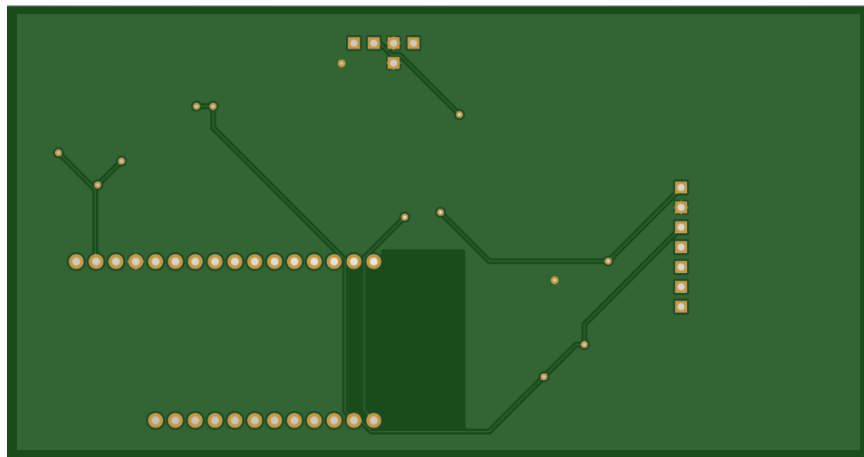
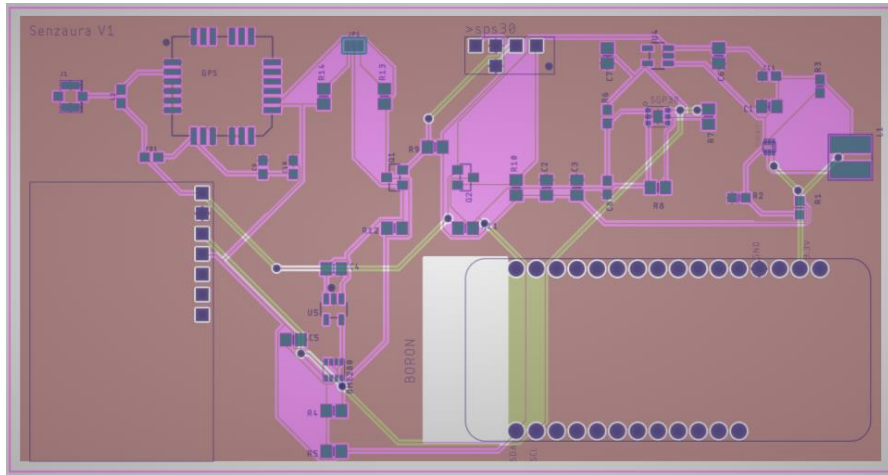
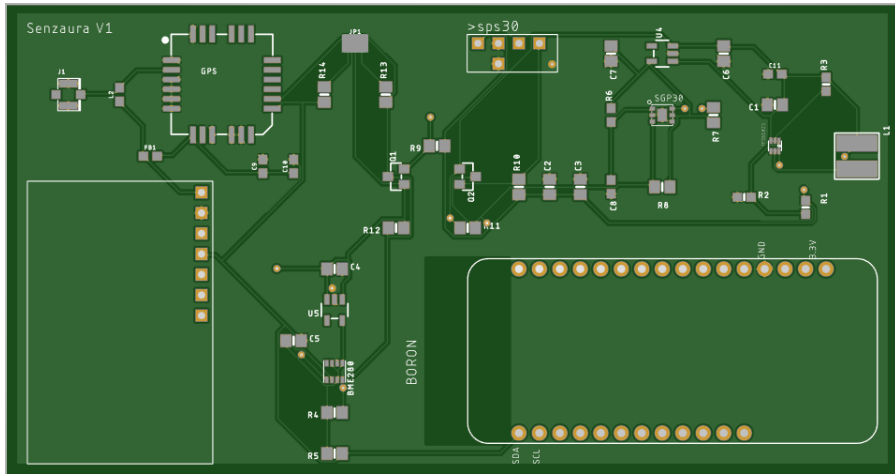


Figure 6.15 Printed Circuit Board Gerber Files Preview

The Parts list and bill of materials are as shown in the following table. The total for 1 unit of our entire system is \$234.34. The battery price can range from \$15-\$30 and it depends on the mAh that is desired by the user. There is no IoT based air quality monitoring device that is this miniaturized that monitors this many parameters.

Designator	Qty	Manufacturer	Mfg Part #	Description / Value	Unit Price
J1	1	SparkFun	WRL-09144	SparkFun Accessories U.FL SMD Connector	\$2.919
L2	1	Würth Elektronik	7847806330	Fixed Inductors WE-MCI 33nH 0.55 Ohms Q-Factor=12	\$0.174
FB1	1	Murata	BLM18KG300WH1D	Ferrite Beads FB SMD 0603inch 30ohm POWRTRN	\$0.330
C8,C9	2	Taiyo Yuden	LMK063C6104KP-F	Multilayer Ceramic Capacitors MLCC - SMD/SMT 0201 10VDC 0.1uF 10% X6S	\$0.105
C10,C11	2	Taiyo Yuden	JMK063ABJ105KP-F	Multilayer Ceramic Capacitors MLCC - SMD/SMT 0201 6.3VDC 1uF 10% X5R	\$0.061
R13,R14	2	Vishay / Dale	CRCW08054K70FKEAC	Thick Film Resistors - SMD 1/8Watt 4.7Kohms 1% Commercial Use	\$0.058
Q1,Q2	2	Onsemi/Fairchild	BSS138	MOSFET SOT-23 N-CH LOGIC	\$0.415
U5	1	Microchip Techn	MIC5225-3.3YM5-TR	LDO Voltage Regulators High Vin, Low Iq Regulator	
BME280	1	Bosch Sensor	BME280	Board Mount Humidity Sensors MEMS humidity, pressure and temperature	\$7.119
C4,C5,C6,C7	4	Taiyo Yuden	LMR212BD7106KG-T	Multilayer Ceramic Capacitors MLCC - SMD/SMT 0805 10VDC 10uF 10% X7T AEC-Q200	\$0.343
GPS	1	Sierra Wireless	XA1110_1104308	Navigation GPS, GLONASS Transceiver Module 1.598GHz ~	\$26.617
SGP30	1	Sensirion AG	SGP30-2.5K	AIR QUALITY GAS SENSOR FOR VOC'S	\$6.825
U4	1	Microchip Techn	TC2015-1.8VCTTR	LDO Voltage Regulators .1mA w/Shtdn & Ref B 1.8V	\$0.960
TPS61023	1	Texas Instrumen	TPS61023DRLR	3.7-A Boost Converter with 0.5-V Ultra-low Input Voltage	\$4.620
L1	1	Würth Elektronik	74438356010	FIXED IND 1UH 7.2A 15 MOHM SMD	\$2.729
R1	1	Vishay / Dale	CRCW0603732KFKEA	Thick Film Resistors - SMD 1/10watt 732Kohms 1%	\$0.123
R2,R3	2	Panasonic	ERJ-H3ED1003V	Thick Film Resistors - SMD 0603 100KOhm 0.5% AEC-Q200	\$0.184
R6	1	Panasonic	ERJ-H3QD4R70V	Thick Film Resistors - SMD 0603 4.70hm 0.5% AEC-Q200	\$0.246
C1,C2,C3	3	Murata	GCM21BD70G226ME36	Multilayer Ceramic Capacitors MLCC - SMD/SMT 22UF 4V 20% 0805	\$0.336
R4,R5,R7,R8	8	Vishay / Dale	RCC080510K0FKEA	Thick Film Resistors - SMD 1/4W 10Kohms 1% 100ppm	\$0.193
Boron	1	Particle	Particle Boron	Microcontroller	\$67.210
SPS30	1	Sensirion AG	SPS30	Particulate Matter Sensor	\$50.240
SCD30	1	Sensirion AG	SCD30	CO2 Sensor	\$62.520
				All Total	\$234.327

Table 6.2 Bill of Materials of System

CHAPTER SEVEN SUMMARY AND FUTURE PLANS

7.1 Summary of Work

The quality of the air we breathe is becoming an increasing issue that needs to be addressed. Infrastructure that monitors and informs people of the quality of their environments needs to be put into place. Internet of Thing devices are a great option to solve this issue. In this research, a fully operational miniaturized IoT Air quality monitoring device was developed. All the sensors chosen were tested and proven to detect the intended air quality monitoring parameters. Then a breadboard-based prototype was built and programmed to fuse all the sensors and allow the data to be visualized in real time from a cloud-based software. After verification, the miniaturization of this system was performed by designing our own proprietary circuits to connect all the components on a printed circuit board. This system was then tested determining its reliability overall performance.

7.2 Possible Future Updates

In the future, sensors could be added to the design as there is available power to supply several more devices to detect different air quality parameters. A casing could be designed to house this system for possible future commercial introduction. A future development that enhances the intended application would be an android or apple based mobile app that would send notifications when a poor or severe air quality condition is detected during operation. If these additions were made, then conducting a study with a

Health System or professional organization to determine the effectiveness of the system would be the next step.

APPENDICES

Appendix A

BME280 SOFTWARE

```
#include <Adafruit_BME280.h>
Adafruit_BME280 bme; //Select I2C for the BME280 (temp, pressure, humidity)
int led = D7;
bool t = false;
bool use_data = true;
void setup() {
  //Particle.function("sled",spiT);
  pinMode(led,OUTPUT);
  digitalWrite(led,HIGH);
  Particle.function("SLed",spiT);
  Serial.begin(9600);
  delay(2000);
  print("BME280 test");
  if (!bme.begin(0x76)) {
    print("Could not find a valid BME280 sensor, check wiring!");
    t=true;
    digitalWrite(led,LOW);
  }
  else {
    print("Found Connection: Starting");
    digitalWrite(led,HIGH);
  }
}
char data[64];
float d[3];
void loop() {
  if( t == false ) {
    d[0] = bme.readTemperature()*9.0/5.0 + 32.0;
    d[1] = bme.readPressure()/101325.0F;
    d[2] = bme.readHumidity();
    sprintf(data, "%0.2f,%0.2f,%0.2f,%0.2f",d[0],[d[0]-32]*5/9,d[1],d[2]);
    Particle.publish("Temp_F,Temp_C,Pressure,Humidity",data);
  }
  else { print("error"); }
  // wait 2 seconds
  delay(2000);
}
void print( String cmd ) {
  if( use_data ) { Particle.publish(cmd); }
  else { Serial.println(cmd); }
}
int spiT(String command) { digitalWrite(led, digitalRead(led)^1 ); return digitalRead(led); }
```

Appendix B

SCD30 SOFTWARE

```
#include <Wire.h>
#include "SparkFun_SCD30_Arduino_Library.h"

int led = D7;
float co2_d[3];

void setup() {
  pinMode(led,OUTPUT);
  digitalWrite(led,HIGH);
  Particle.function("BME_Led",BMET);
  Wire.begin();
  Serial.begin(9600);
  Serial.println("SCD30 Example");
  SCD30.begin()
}

void loop() {
  if (SCD30.dataAvailable()) {
    co2_d[0] = SCD30.getTemperature()*(9.0/5.0) + 32;
    co2_d[1] = SCD30.getCO2();
    co2_d[2] = SCD30.getHumidity();

    Serial.print("co2(ppm):");
    Serial.print(co2_d[1]);
    Serial.print(" temp(F):");
    Serial.print(co2_d[0], 1);
    Serial.print(" temp(C):");
    Serial.print( (co2_d[0]-32)*(5.0/9.0), 1);
    Serial.print(" humidity(%):");
    Serial.print(co2_d[2], 1);
    Serial.println();
  }
  else { Serial.println("NA"); }

  delay(2000);
}

Int BMET(String command) {
  digitalWrite(led, digitalRead(led)^1 );
  return digitalRead(led);
}
```

Appendix C
SPS30 SOFTWARE

```
#include "SPS30.h"

SPS30 Sensor;

int led = D7;

void setup() {

  pinMode(led,OUTPUT);
  digitalWrite(led,HIGH);
  Particle.function("PM-Led",pmT);

  Wire.begin();

  Serial.begin(9600);
  Serial.println("SPS30 Example");

  Sensor.begin();

  if( !Sensor.begin() ) {
    Particle.publish("SENSOR NOT DETECTED");
    delay(500);
    // System.reset();
  }
}

float mass_concen[4];
float num_concen[5];

char *pm[5] = {"PM0.5", "PM1.0", "PM2.5", "PM4.0", "PM10"};

int i=0;

void loop() {
  if (Sensor.dataAvailable()) {
    Sensor.getMass(mass_concen);
    Sensor.getNum(num_concen);

    char data[4];

    Particle.publish("--Mass Concentration--");
```

```

for(i=0; i<4;i++) {
  sprintf(data, "%s: %0.2f ", pm[i+1], mass_concen[i]);
  Particle.publish("", data);
  // Serial.printf("%s: %0.2f\n", pm[i+1],mass_concen[i]);
}

Serial.println("--Number Concentration--");
for(i=0; i<5;i++) {
  Serial.printf("%s: %0.2f\n", pm[i],num_concen[i]);
}

}
else { Particle.publish("NA"); }
delay(2000);
}

int pmT(String command) {
  digitalWrite(led, digitalRead(led)^1 );
  return digitalRead(led);
}

```

Appendix D

SGP30 SOFTWARE

```
#include "Adafruit_SGP30.h"
#include <Wire.h>
#include <math.h>
#include <Particle.h>

Adafruit_SGP30 sgp;

uint32_t getAbsoluteHumidity(float temperature, float humidity) {
    // approximation formula from Sensirion SGP30 Driver Integration chapter 3.15
    const float absoluteHumidity = 216.7f * ((humidity / 100.0f) * 6.112f * exp((17.62f * temperature) /
(243.12f + temperature)) / (273.15f + temperature)); // [g/m^3]
    const uint32_t absoluteHumidityScaled = static_cast<uint32_t>(1000.0f * absoluteHumidity); //
[mg/m^3]
    return absoluteHumidityScaled;
}

void setup() {
    Wire.begin();
    Serial.begin(115200);
    while (!Serial) { delay(10000); } // Wait for serial console to open!

    Particle.publish("SGP30 test");

    if (!sgp.begin()){
        Particle.publish("Sensor not found :(");
        delay(10000);
    }

    if (!sgp.begin()){
        Particle.publish("Sensor not found :(");
        delay(10000);
    }

    if (!sgp.begin()){
        Particle.publish("Sensor not found :(");
        delay(10000);
    }

    Particle.publish("Found SGP30 serial #");
    Serial.print(sgp.serialnumber[0], HEX);
    Serial.print(sgp.serialnumber[1], HEX);
    Serial.println(sgp.serialnumber[2], HEX);
```



```

}

int counter = 0;
void loop() {

  if (!sgp.IAQmeasure()) {
    Particle.publish("Measurement failed");
  }
  Particle.publish("working");
  Particle.publish("TVOC", String(sgp.TVOC));
  delay(10000);

  counter++;
  if (counter == 30) {
    counter = 0;

    uint16_t TVOC_base, eCO2_base;
    size_t readBytes(char*TVOC_base,size_t length);
    if (!sgp.getIAQBaseline(&eCO2_base, &TVOC_base)) {
      Serial.println("Failed to get baseline readings");
      return;
    }
    Particle.publish("TVOC base",String(TVOC_base,HEX));
    Serial.print("****Baseline values: eCO2: 0x"); Serial.print(eCO2_base, HEX);
    Serial.print(" & TVOC: 0x"); Serial.println(TVOC_base, HEX);

  }
}

```

Appendix F
GPS SOFTWARE

```
#include "TinyGPS.h"
#include "SparkFun_I2C_GPS.h"

I2CGPS myI2CGPS;
TinyGPSPlus gps;
bool GPS_connected = true;
unsigned long msDelay=0;
double lat,lng;
bool rst=false;

void setup() {
  Serial.begin(115200);
  Serial.println("GTOP Read Example");
  Particle.function("RESET",RST_DVC);
  Particle.variable("Latitude",lat);
  Particle.variable("Longitude",lng);
  if (myI2CGPS.begin() == false) {
    Serial.println("Module failed to respond. Please check wiring.");
    while (1); //Freeze!
  }
  Serial.println("GPS module found!");
}

void loop() {
  if( millis() - msDelay < 1*60*1000 ) {return;}
  msDelay = millis();
  while (myI2CGPS.available()) {
    gps.encode(myI2CGPS.read()); //Feed the GPS parser
  }
  if (gps.time.isUpdated()) {
    displayInfo();
  }
}
char data[200];
//Display new GPS info
void displayInfo() {
  Serial.println();
  if (gps.time.isValid()) {
    Serial.print(F("Date: "));
    Serial.print(gps.date.month());
    Serial.print(F("/"));
    Serial.print(gps.date.day());
```

```

Serial.print(F("/"));
Serial.print(gps.date.year());

Serial.print((" Time: "));
if (gps.time.hour() < 10) Serial.print(F("0"));
Serial.print(gps.time.hour());
Serial.print(F(":"));
if (gps.time.minute() < 10) Serial.print(F("0"));
Serial.print(gps.time.minute());
Serial.print(F(":"));
if (gps.time.second() < 10) Serial.print(F("0"));
Serial.print(gps.time.second());
Serial.println(); //Done printing time

sprintf(data, "Date: %d/%d/%d - Time:
%2d:%2d:%2d",gps.date.month(),gps.date.day(),gps.date.year(),gps.time.hour()+(int)round(gps.location.
lng() /20),gps.time.minute(),gps.time.second() );
Particle.publish("Data&Time",data);
}
else {
Serial.println(F("Time not yet valid"));
Particle.publish("Date&Time", "Time not yet valid");
}

if (gps.location.isValid()) {
Serial.print("Location: ");
Serial.print(gps.location.lat(), 6);
Serial.print(F(", "));
Serial.print(gps.location.lng(), 6);
Serial.println();

lat = (double) gps.location.lat();
lng = (double) gps.location.lng();

sprintf(data, "%2.7f,%2.7f",lat,lng);
Particle.publish("GPS",data);
}
else {
Serial.println(F("Location not yet valid"));
Particle.publish("Location", "Location not yet valid");
}
}

int RST_DVC(String command) { rst = true; return rst; }

```

Appendix G

FUSED SENSOR SYSTEM SOFTWARE

```
// This #include statement was automatically added by the Particle IDE.
#include "Adafruit_SGP30.h"
#include "SPS30.h"
#include "SparkFun_I2C_GPS.h"
#include "TinyGPS.h"
#include <Adafruit_BME280.h>
#include <SparkFun_SCD30_Arduino_Library.h>
#include <Wire.h>
#include <math.h>
#include <Particle.h>

SPS30 SPS;      // Create the SPS30 module
Adafruit_SGP30 sgp; // create SGP30 module
Adafruit_BME280 bme; //Select I2C for the BME280 (temp, pressure, humidity)
SCD30 SCD;      // Create the SCD30 module
I2CGPS myI2CGPS; // Create the GPS module
TinyGPSPlus gps; // Create the GPS interpreter
STARTUP(System.enableFeature(FEATURE_RETAINED_MEMORY)); //Allows the system to retain
variables for quick startup
retained int t_zone = -4; // set the default time zone to -4 (east coast USA) with saving ability

// main blue LED for use in status signals and connecting checking
int led = D7;

// used later in the functions
#define temp_F 0
#define temp_C 1
#define pressure 2
#define humidity 3
#define CO2 4
#define VOC 5
#define cPM1 6
#define cPM2_5 7
#define cPM4 8
#define cPM10 9

// used to hold raw data from all sensors. Temporary initialized
retained double data[10] = {-1,-1,-1,-1,-1,-1,0.401,0.425,0.440,0.410};

// GPS saved values
retained double lat = 0.0;
retained double lng = 0.0;
```

```

// variable used later in the function
unsigned long msDelay=0; //last ms count at data send
unsigned long gpsDelay=0; //last ms count at GPS update
int past; // the day when the system started
bool rst = false; // resetting variable for user quick reset
bool slp = false; // sleeping variable for user full reset

// hold the values for SPS sensor
float mass_concen[4];
//Average Temp in Celcius
float Ctemp_avg;

// Connection bits
int SGP_connected = 0; // new SGP30 connection bit
int BME_connected = 0; // the BME280 connection bit
int SCD_connected = 0; // the SCD30 (Sensirion CO2) connection bit
int SPS_connected = 0; // the SPS30 (Sensirion PM) connection bit
int GPS_connected = 0; // the GPS connection bit
int GPS_count=8;

// GPS Testing Variable
retained int gps_time = 60;
retained int sense_time = 30;

void setup() {
  //setting time zone to user selected time zone (default: -4)
  Time.zone(t_zone);

  Wire.begin();
  Serial.begin(115200);

  while (!Serial) {
    delay(1000);
  }

  // Connecting to the Particulate Matter sensor
  if(!SPS.begin())
    Particle.publish("SPS30", "Could not found a valid SPS30!");
  else
    SPS_connected = 1;

  // Connecting to the VOC sensor
  if(!sgp.begin())
    Particle.publish("SGP", "Could not found a valid SGP!");

```

```

else
  SGP_connected = 1;

// Connecting to the Temp, pressure, humidity sensor
if (!bme.begin(0x77))
  Particle.publish("BME280","Could not find a valid BME280!");
else
  BME_connected = 1;

// Connecting to the CO2 sensor
if (!SCD.begin())
  Particle.publish("SCD30","Could not find a valid SCD30!");
else
  SCD_connected = 1;

// Connecting to the GPS module
if (!myI2CGPS.begin())
  Particle.publish("GPS","Could not find the GPS module!");
else
  GPS_connected = 1;

// Update the GPS location if the GPS is ready
gpsUpdate();

Particle.publish("STARTING","Starting the data aquisition!");

// Save starting day
past = Time.day();
}

void loop() {
  // reset once everyday at the begining of the day
  DailyReset();

  // check to see if GPS needs updating (every 1 minute)
  if( millis() - gpsDelay > 1000*gps_time ) {
    gpsDelay = millis(); gpsUpdate();
  }
  //If the system has been running to long the millis() can reset to 0
  if( millis() < gpsDelay ) {
    gpsDelay=millis();
  }

  // toggle every 15 seconds
  if(millis() - msDelay > 500*sense_time) {

```

```

    ledToggle("");
}

// reset if it hasnt been 30 seconds
if(millis() - msDelay < 1000*sense_time) {
    return;
}
// if time has looped to high, reset timer
else if( millis() < msDelay) {
    msDelay=millis();
}
msDelay = millis(); // save time if it has been 30 seconds

// fuction call to read data of various sensors
getSps30();
getSgp();
getBme280();
getScd30();

// formatted string output
String out =
String::format("\temp\":%.1f,\"HMD\":%.1f,\"press\":%.1f,\"CO2\":%.0f,\"VOC\":%.0f,\"cPM1\":%.2
f,\"cPM2_5\":%.2f,\"cPM4\":%.2f,\"cPM10\":%.2f)",
                data[temp_F], data[humidity], data[pressure], data[CO2], data[VOC], data[cPM1],
data[cPM2_5], data[cPM4], data[cPM10]);

Particle.publish("blob_full", out, PRIVATE, WITH_ACK);
delay(10000);
}

// fuction defination
void getSps30() {
    if( SPS.dataAvailable()) {
        SPS.getMass(mass_concen);
        data[cPM1] = mass_concen[0];
        data[cPM2_5] = mass_concen[1];
        data[cPM4] = mass_concen[2];
        data[cPM10] = mass_concen[3];
    }
}

void getSgp() {
    if (sgp.IAQmeasure()) {
        data[VOC] = sgp.TVOC;
    }
}

```

```

void getBme280() {
  if(BME_connected) {
    data[temp_C] = toC( (110.0/125.0) * toF(bme.readTemperature()) );
    data[pressure] = (double)bme.readPressure()/100;
    data[humidity] = bme.readHumidity()+10;
  }
}

void getScd30() {
  if(SCD_connected) {
    if(SCD.dataAvailable()) {
      data[temp_F] = (110.0/125.0) * toF(SCD.getTemperature());
      data[CO2] = (1.125) * (double)SCD.getCO2();
    }
  }
}

void TempAvg() {
  if(SCD_connected && BME_connected) {
    Ctemp_avg = ( data[temp_C] + toC(data[temp_F]) ) /2;
    data[temp_C] = Ctemp_avg;
    data[temp_F] = toF(Ctemp_avg);
  }
  else if ( SCD_connected ) { data[temp_C] = toC(data[temp_F]); }
  else if (BME_connected) { data[temp_F] = toF(data[temp_C]); }
}

bool range=true;
double value=0.0001;

int gpsUpdate() {
  if(GPS_connected) { //if the GPS is connected...
    // read the GPS and encode it into the parser
    while( myI2CGPS.available() ) { gps.encode(myI2CGPS.read()); }
    if( gps.time.isUpdated() ) { //If the GPS has sent new data
      if( gps.location.isValid() ) { // and the data is valid
        lat = (double) gps.location.lat(); // get the latitude
        lng = (double) gps.location.lng(); // get the longitude
        char g[50]; sprintf(g,"%2.5f,%2.5f",lat,lng); // create the data packet
        Particle.publish("GPS",g,PRIVATE,WITH_ACK); // send the data packet
        GPS_count=0;
        return 0; //return success
      }
    }
    else { // if GPS has sent data but its not valid
      GPS_count++;
    }
  }
}

```



```

Particle.publish("Invalid","GPS location not yet valid");
if(GPS_count>4) {
  GPS_count=0;
  if(lat!=0 && lng!=0) {
    value = (double)((range)?0.00001:0.0);
    char g[50]; sprintf(g,"%2.5f,%2.5f",lat+value,lng); // create the data packet
    Particle.publish("GPS",g,PRIVATE,WITH_ACK); // send the data packet
    range=!range;
  }
}
return 0;
}
}
return 1; // if the GPS is not connected or the data has not updated then return failure
}

void DaylyReset() {
  if( Time.day() != past ){
    // print the message
    Particle.publish("NORM_RST","The normal daily reset (will last 15 minutes)");
    Wire.end(); // stop I2C system
    delay(200); // wait for message to send and I2C system to stop
    System.sleep(SLEEP_MODE_DEEP, 900); //sleep for 15 minutes
  }
}

// helper functions
// convert to Farenheit
float toF(float value) {
  return value*(9.0/5.0) + 32.0;
}

// convert to Celcius
float toC(float value) {
  return (5.0/9.0)*(value-32.0);
}

// toggle the built in LED
int ledToggle(String command) {
  digitalWrite(led, digitalRead(led)^1);
  return digitalRead(led);
}

// set the UTC zone
int utcSet(String command) {

```

```
    t_zone = atoi(command); return t_zone;
}
```

```
//retrieve the previously set UTC zone
int utcGet(String command) {
    return t_zone;
}
```

```
int gpsSet(String command) {
    int value = atoi(command);
    if( value >= 1 ) {
        gps_time = value;
    }
    else {
        gps_time = 60;
    }
    return gps_time;
}
```

```
int senseSet(String command) {
    int value = atoi(command);
    if( value >= 5 ) {
        sense_time = value;
    }
    else {
        sense_time = 30;
    }
    int n = 100*(10*sense_time/31);

    return sense_time + 100*n;
}
```

Appendix L
MICROPROCESSOR DATASHEET

Particle Boron:

<https://docs.particle.io/reference/datasheets/b-series/boron-datasheet/>

Appendix M
SENSOR DATA SHEETS

SGP30:

https://cdn-learn.adafruit.com/assets/assets/000/050/058/original/Sensirion_Gas_Sensors_SGP30_Datasheet_EN.pdf

SCD30:

https://sensirion.com/media/documents/4EAF6AF8/61652C3C/Sensirion_CO2_Sensors_SCD30_Datasheet.pdf

SPS30:

https://www.digikey.com/htmldatasheets/production/3483760/0/0/1/sps30.html?utm_adgroup=General&utm_source=google&utm_medium=cpc&utm_campaign=Dynamic%20Search_EN_RLSA_Cart&utm_term=&utm_content=General&gclid=Cj0KCQjw2MWVBhCQARIsAIjbwoM9oylSFwPn_akiK_flQX5NSTbpA8xqCAaj_MhE8nbdONh0RmrvbwlIaAr7EEALw_wcB

BME280:

<https://www.bosch-sensortec.com/media/boschsensortec/downloads/datasheets/bst-bme280-ds002.pdf>

XA1110:

https://media.digikey.com/pdf/Data%20Sheets/Sierra%20Wireless%20PDFs/AirPrime_XA1110_TechSpec_Rev1_6-23-17.pdf

Pull up Resistors:

https://www.ti.com/lit/an/slva485/slva485.pdf?ts=1657408509966&ref_url=https%253A%252F%252Fwww.ti.com%252Fsitesearch%252Fenus%252Fdocs%252Funiversalsearch.tsp%253FlangPref%253Den-US%2526searchTerm%253Dcalculate%2Bpull%2Bup%2Bresistor%2Bvalue%2526nr%253D12918

REFERENCES:

- [1] “Air pollution,” *World Health Organization*, 24-Sep-2019. [Online]. Available: <https://www.who.int/airpollution/en/>. [Accessed: 20-June-2022].
- [2] “Standards and Guidelines,” *American Society of Heating, Refrigerating, and Air Conditioning Engineers*. [Online]. Available: <https://www.ashrae.org/technical-resources/standards-and-guidelines>. [Accessed: 07-June-2022].
- [3] “Mold Prevention Strategies and Possible Health Effects in the Aftermath of Hurricanes and Major Floods,” *Center of Disease Control*. [Online]. Available: <https://www.cdc.gov/mmwr/preview/mmwrhtml/rr5508a1.htm> [Accessed: 21-June-2022].
- [4] “Associations of Cognitive Function Scores with Carbon Dioxide, Ventilation, and Volatile Organic Compound Exposures in Office Workers: A Controlled Exposure Study of Green and Conventional Office Environments,” *Environmental Health Perspectives*. [Online]. Available: <https://ehp.niehs.nih.gov/doi/10.1289/ehp.1510037>. [Accessed: 04-May-2022].
- [5] “How we calculate our air quality index and why we need it,” *Breeze Technologies*. [Online]. Available: <https://www.breeze-technologies.de/blog/what-is-an-air-quality-index-how-is-it-calculated/> [Accessed: 04-July-2022].
- [6] “Warning Signs and Symptoms of Heat-Related Illness,” *Center for Disease Control and Prevention*. [Online]. Available:

- <https://www.cdc.gov/disasters/extremeheat/warning.html> [Accessed: 01-July-2022].
- [7] Pahlevi, R. R. *Fast UART and SPI Protocol for Scalable IoT Platform*. 2018, pp. 239–44, <https://doi.org/10.1109/ICoICT.2018.8528745>.
- [8] Houghton, William. *Advantages of I2c Protocol for Microcontroller Applications*. 1991, pp. 22–27, <https://doi.org/10.1109/ELECTR.1991.718167>.
- [9] “Basics of UART Communication,” *Circuit Basics*, 11-Apr-2017. [Online]. Available: <http://www.circuitbasics.com/basics-uart-communication/>. [Accessed: 10-May-2022].
- [10] “Basics of the SPI Communication Protocol,” *Circuit Basics*, 23-May-2018. [Online]. Available: <http://www.circuitbasics.com/basics-of-the-spi-communication-protocol>. [Accessed: 10-May-2022].
- [11] “Basics of the I2C Communication Protocol,” *Circuit Basics*, 11-Apr-2017. [Online]. Available: <http://www.circuitbasics.com/basics-of-the-i2c-communication-protocol>. [Accessed: 10-May-2022].
- [12] “This is what dirty air does to your body,” *Share America*, 10-Nov-2015. [Online]. Available: <https://share.america.gov/this-is-what-dirty-air-does-to-your-body/> [Accessed: 18-June-2022].