Sulav Thapa & Subash Chandra K.C.

Raspberry Pi and ESP32-Based Smart Sensor Network for IoT Platform Integration and Real-Time Environmental Data Monitoring

Metropolia University of Applied Sciences

Bachelor of Engineering

Information Technology

Bachelor's Thesis

30 November 2023

Abstract

Author:	Sulav Thapa & Subash Chandra K.C.
Title:	Raspberry Pi and ESP32-Based Smart Sensor Network for IoT Platform Integration and Real-Time Environmental Data Monitoring
Number of Pages:	40 pages + 1 appendix
Date:	30 November 2023
Degree:	Bachelor of Engineering
Degree Programme:	Information Technology
Professional Major:	Smart IoT Systems
Supervisors:	Erik Pätynen, Senior Lecturer

The aim of this final year project was to develop two sets of identical ESP32 based field devices which would be able to get data from different environmental sensors such as a temperature and humidity sensor, a lux sensor, a barometric pressure and altimeter sensor and an air quality sensor and publish the data to cloud services. A main control unit was also developed which was to subscribe the published data from field devices and store it locally as well as publish them in the IoT cloud. Subsequently the designed system was tested by placing the first ESP32 based field device in a different location from the second ESP32 based field device which then would send data to the main device in a totally different location.

In order to achieve the goals, all of the environmental sensors were connected individually to ESP32, tested for the data and connected to the AWS cloud for transmitting data. The same data was acquired using a Raspberry Pi as a main control unit which would collect data from all the field devices, create a text file for a single day, and append the data. Furthermore, if a great amount of data would arrive at the same time frame from different field devices, all the data would be averaged and appended along the other logged data. Also, the resulting data would be pushed to an IoT cloud platform called Blynk.

The proposed model was tested keeping the two identical field devices in different locations, collecting data sending it to cloud and the main device in a totally different place collecting, analyzing and publishing data. The data was collected for a whole day and analyzed based on all the environmental data collected. It can be concluded that the cheap alternative sensing and monitoring system that was experimented with in this project with a master-slave architecture, where the master can collect data from several slaves or field devices, can be practiced in a real-life scenario.

Keywords: IoT system, AWS cloud, Raspberry Pi.

Contents

List of Abbreviations

1 Introduction 1

2 Technical Background 3

2.1 System Design 3
2.2 Hardware Selection 6
2.2.1 Raspberry Pi 4 6
2.2.2 ESP32-S3 8
2.2.3 SHTC3 (Temperature and Humidity Sensor) 9
2.2.4 BMP390 (Precision Barometric Pressure & Altimeter Sensor) 10
2.2.5 VEML7700 (Lux Sensor) 11
2.2.6 SGP30 (Sensor for Air Quality) 12
2.2.7 Minor Components 13
2.3 Software Selection 13
2.3.1 Raspberry Pi OS 13
2.3.2 AWS IoT Core 13
2.3.3 Python3 13
2.3.4 Python3 Libraries 14
2.3.5 PlatformIO 14
2.3.6 PlatformIO Libraries 14

3 System Implementation 16

4 Results 29

5 Conclusions 35

References 37

Appendices
Appendix 1: Source Code

List of Abbreviations

AC: Alternating current

AWS: Amazon web services

DC: Direct current

eCO2: Equivalent carbon dioxide reading

GPIO: General-purpose input/output

I2C: Inter-integrated circuit

IOT: Internet of things

JSON: JavaScript object notation

LAN: Local area network

MCU: Microcontroller unit

MQTT: Message queuing telemetry transport

NTP: Network time protocol

PSRAM: Pseudo static random-access memory

PVC: Polyvinyl chloride

RAM: Random access memory

SPI: Serial peripheral interface

SRAM: Static random-access memory

TVOCs: Total volatile organic compounds

USB: Universal serial bus

VS: Visual studio

VOCs: Volatile organic compounds

1 Introduction

Most designed systems have a processing unit on board and several sensors connected to the same unit. These units are basically a single system that is responsible for sensing the environmental or any other required data from the sensors and pushing the data to a server. In case of any failure the whole system stays dormant and the data are not received on the server end.

Big industrial plants use monitoring systems where environmental data such as temperature, humidity, air quality index, pollution index, carbon dioxide concentration, atmospheric pressure, light intensity, and volatile organic compounds (VOCs) are to be monitored and in case of any threat caused by these parameters, notifications or warnings are triggered. Usually, industrial grade sensing and monitoring systems are centralized, very expensive and high tech.

This final year project experiments with the use of a cheap alternative to environmental data sensing and monitoring systems with a very different architecture in mind. This thesis focuses on mainly master-slave architecture where a master can collect data from several slaves or field devices. Big cities, big greenhouses and large industries require many data setpoints in order to summarize the environmental data of the whole entity. These smaller, inexpensive, affordable systems can be totally isolated from the main master devices while still sending data to the main master devices in real time. The failure of a slave device would have no impact on the other field devices or even the master device. Automation and embedded solutions are advancing towards a great future. Wi-Fi is available everywhere these days with an easy access to the internet. Because of the availability of Wi-Fi, the developed solution could be tested in a real case scenario for some period of time. Also, the feasibility of similar devices can be tested in the near future.

This project involved using an ESP32 microcontroller board along with various sensors, including a temperature/humidity sensor, a light sensor, a barometric

pressure sensor, an air quality sensor as a field device and a Raspberry Pi based master device for data retrieval, storage and publishing to an IoT (internet of things) platform for further processing and analysis.

The main objectives of this project included the following:

- To develop two sets of identical ESP32 based field devices which would be able to get data from different environmental sensors such as a precise SHTC3 temperature and humidity sensor, a VEML7700 lux sensor, a precise barometric BMP390 pressure and altimeter sensor, and an SGP30 air quality breakout sensor and to publish the obtained data to cloud services such as AWS (Amazon web services).
- To develop a main control unit based on Raspberry Pi which would be able to subscribe the published data from the field devices and store them locally as well as publish them on the IoT cloud platform for viewing as well as further analysis.
- To test the designed system placing the first ESP32 based field device in a different location from the second ESP32 based field device which would then send data to the main device in a totally different location.

2 Technical Background

The following section will provide a detailed overview of the technical aspect of the system, what hardware were selected, the overall system architecture and how the overall system was designed.

2.1 System Design

Figure 1 illustrates the system design for the ESP32 based field devices. The system consists of four digital sensors to collect environmental data, a microcontroller ESP32 to acquire data from these sensors and a built-in wireless Wi-Fi module to forward the data to the AWS cloud.

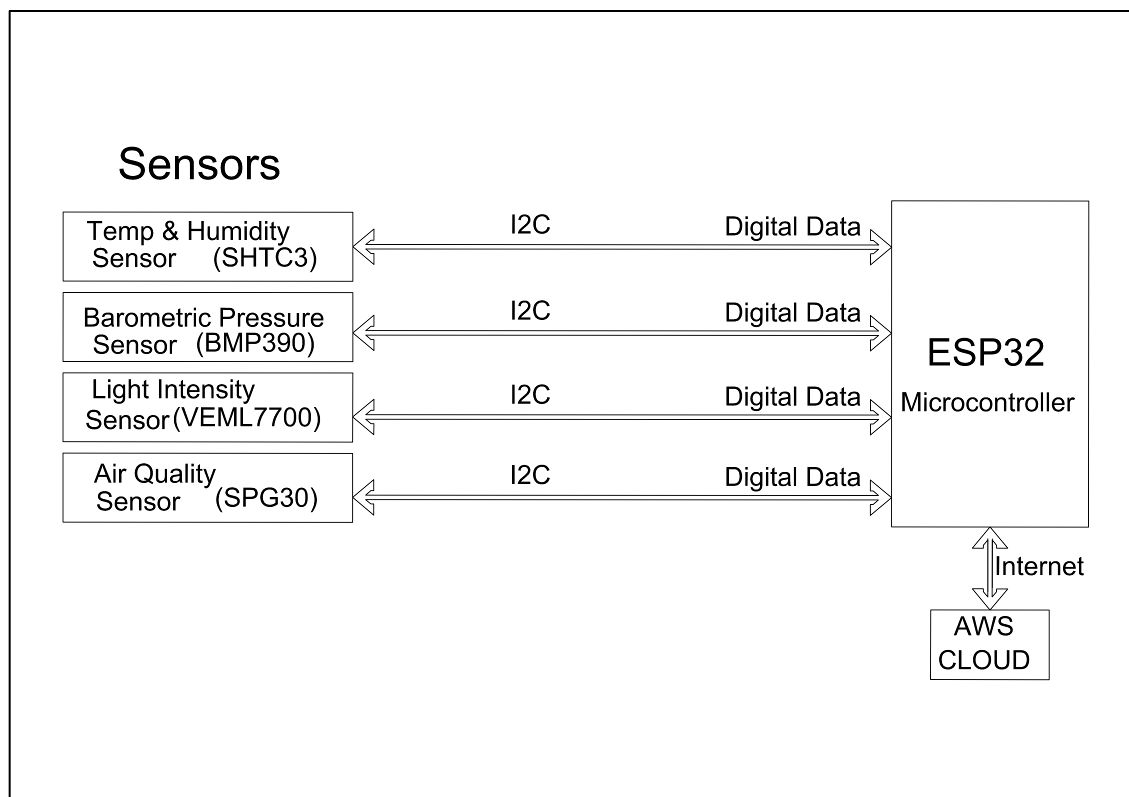

Figure 1. System design of field devices.

The developed solution with ESP32 and sensors are called field devices because these devices have no onboard data storage system and because the data are collected in real-time and sent to the AWS cloud in real-time as well. Field devices

are the hardware devices that are actually located in the field, and many of the devices can be deployed in the field.

The SHTC3 sensor was responsible for collecting air temperature and air humidity data. The BMP390 sensor provides barometric pressure and altitude data. In order to assess air quality in terms of VOCs and eCO2, the SPG30 air quality sensor was employed. Additionally, VEML7700 sensor is used to measure the light intensity within the area in terms of lux.

All of the sensors are I2C (inter-integrated circuit) sensors. All of the sensors are connected to the same I2C bus. All of the sensors are digital and communicate via the I2C bus.

Various libraries were used to process data coming from the sensors. The SparkFun_SHTC3_Arduino_Library library is used to get data from SHTC3 into ESP32. The Adafruit_BMP3XX library is used for BMP390 sensor breakout. The Adafruit_VEML7700 library for the VEML7700 sensor breakout and the Adafruit_SGP30 library for the SGP30 sensor breakout were used in order to get these environmental data in ESP32. All of these libraries are compatible with ESP32. These libraries play a crucial role in turning machine readable data to a human readable format.

ESP32 is a microcontroller so once it turns on the code stored, it will automatically start to run. The code will try to collect data from the sensor and push the collected data to the AWS cloud.

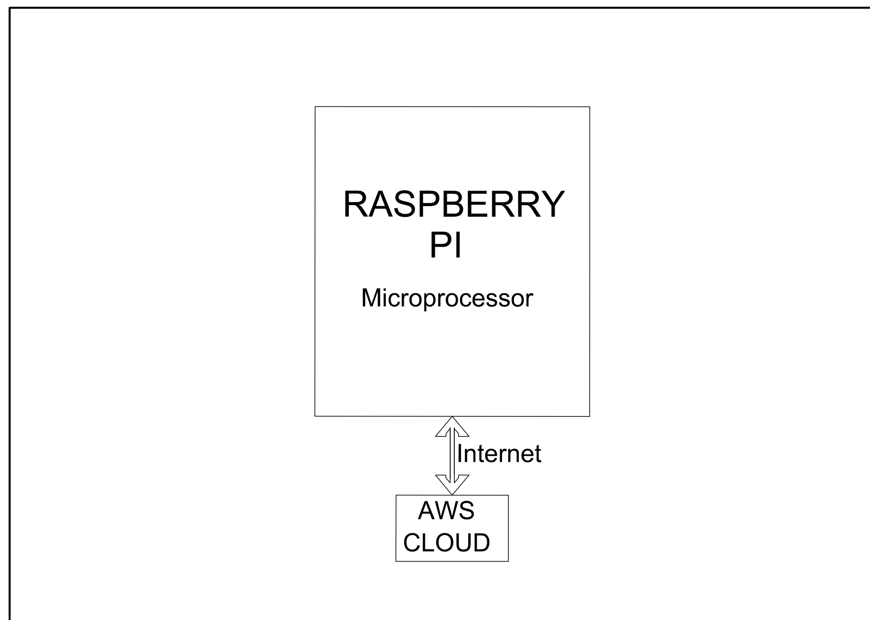

Figure 2. System design of master device.

Figure 2 demonstrates the system diagram for the master device which is just a Raspberry Pi connected to the AWS cloud.

There are already field devices that are collecting the data of the environment. The sensors do not have to be connected to Raspberry Pi. Raspberry Pi is the master device which is not located on the field, but is connected to the internet and subscribed to the AWS cloud.

Once the data are pushed from these field devices, the master device or the main device gets the data. These data coming from the field devices are locally stored in the Raspberry Pi. Then, the data are pushed with a timestamp on them so that the master device could check if the data are of the same time. If they were of the same time, then the data was averaged and finally pushed to the Blynk IoT software platform for viewing as well as further analysis.

The main aim of this project was to develop two sets of identical ESP32 based field devices which will be able to get data from different environmental sensors and publish the data to cloud services such as AWS to develop a main control unit based on Raspberry Pi which will subscribe the published data from field

devices, store them locally, publish them in the IoT cloud platform, and test the designed system placing the first ESP32 based field device in a different location from the the second ESP32 based field device which will then send data to the main device in a totally different location.

2.2 Hardware Selection

Considering these objectives of the project, hardware was selected. The selected hardware are as follows:

- Raspberry Pi 4
- Raspberry Pi 4 power supply
- Raspberry Pi 4 case
- Micro SD card
- Micro SD card reader
- Esp32-s3
- Micro USB (universal serial bus) cable for esp32
- SHTC3 precise temperature and humidity sensor
- VEML7700 lux sensor
- BMP390 precise barometric pressure and altimeter sensor
- SGP30 air quality sensor breakout, VOC and eCO2 sensor
- STEMMA QT / Qwiic JST SH 4-pin cable
- Jumper cable
- Double sided tape
- Power bank
- Water proof box
- PVC (Polyvinyl chloride) tape

2.2.1 Raspberry Pi 4

In 2022, the popular Raspberry Pi series of computers was the Raspberry Pi 4 Model B. It is equipped with a powerful 64-bit quad core processor, which can support up to 4GB of random-access memory There's a dual band 2.4GHz and

5GHz wireless LAN network in the local area. It's got Bluetooth 5.0 and gigabit ethernet as well. It has universal serial bus USB 3.0 for external peripherals. The Raspberry Pi hardware, along with all the main connectors is illustrated in Figure 3 below. [1.]

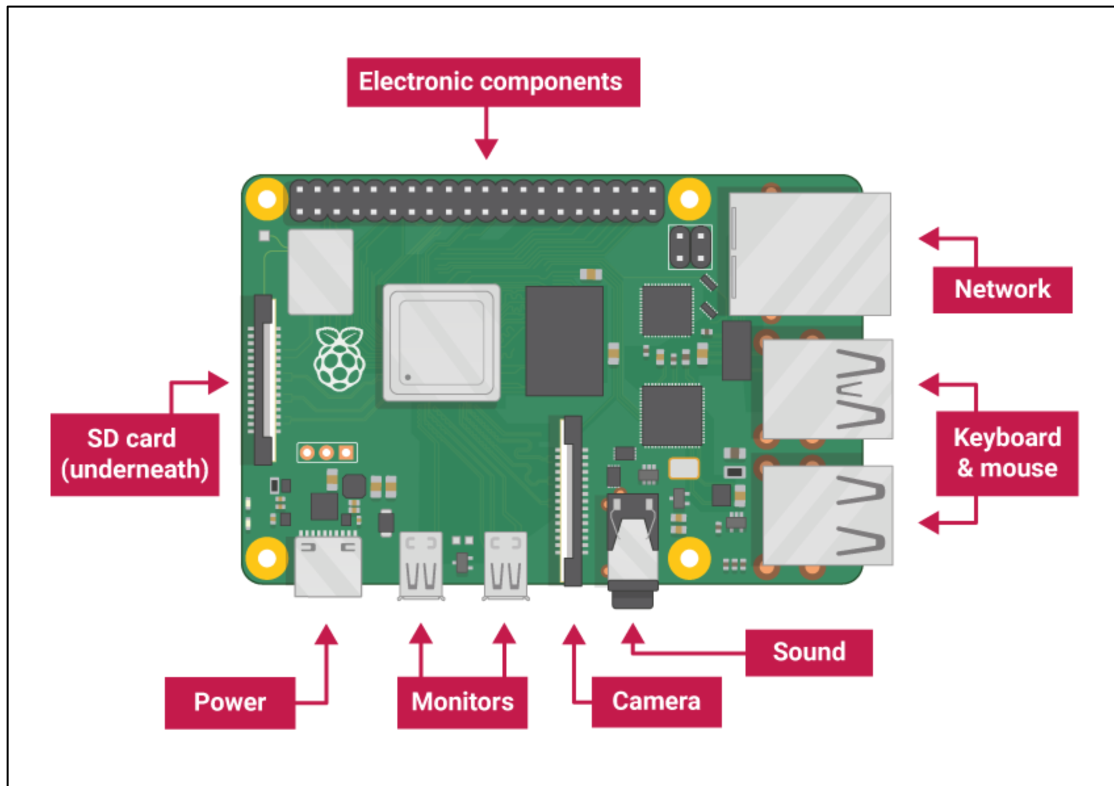

Figure 3. Raspberry Pi with all the peripherals [2].

This project uses wireless LAN to receive the data from the field devices via the AWS IoT cloud platform. Then it uses the processor, the RAM and the memory of the hardware to analyze the collected data and to save the environmental data in local storage. After that wireless LAN was used to push the analyzed data to the Blynk IoT cloud platform for further viewing and analysis if needed.

2.2.2 ESP32-S3

The ESP32S3 MCU, which is capable of operating at 240 MHz using the two core XTENSYSA LX7”), a multicore microcontroller. It's got 512 KB of internal SRAM, and on top of that, it comes with built-in 2.4 GHz, 802.11 b/g/n Wi-Fi, and Bluetooth 5 (LE) for long-range connectivity. With 45 programmable GPIOs and a range of peripherals, ESP32-S3 is designed to support various functionalities. Additionally, it supports a larger, high-speed octal SPI flash and PSRAM with configurable data and instruction cache. Figure 4 below demonstrates the hardware development kit of ESP32-S3 which was used in the project. [3.]

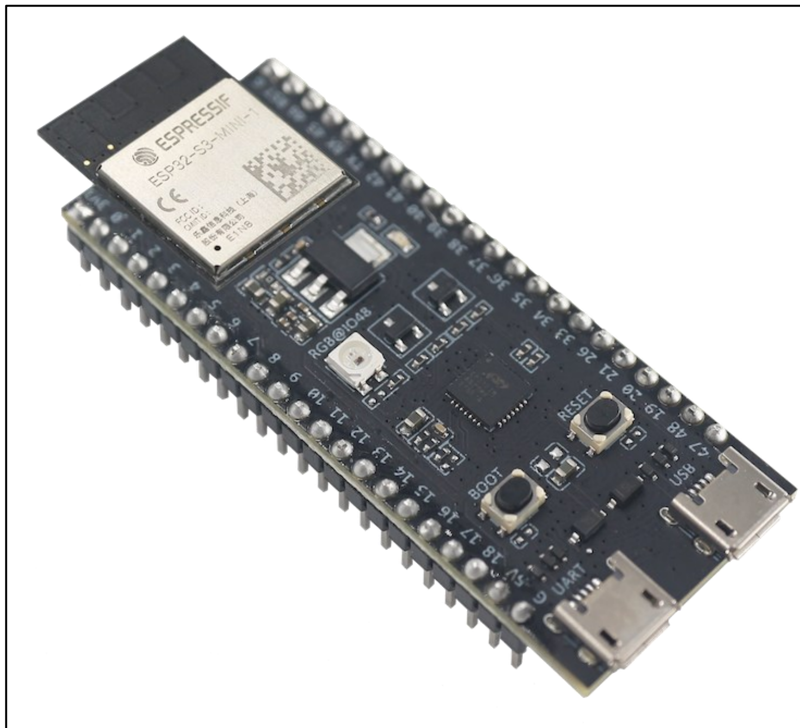

Figure 4. ESP32-S3 development kit [4].

ESP32-S3 development kit is the main processing part of the field devices, and the sensors are connected to GPIOs. The project uses of ESP32 processor and RAM to gather data from all the sensors and uses its wireless connectivity to send these data to the AWS cloud.

2.2.3 SHTC3 (Temperature and Humidity Sensor)

Some of the best and most accurate temperature and humidity sensors are SHT temperature and humidity sensors. The SHTC3 sensor has excellent ±2% relative humidity and ±0.2 °C accuracy. This sensor's address 0x70 has a I2C interface. Figure 5 below demonstrates the breakout development board for SHTC3 which was used in the project. [5.]

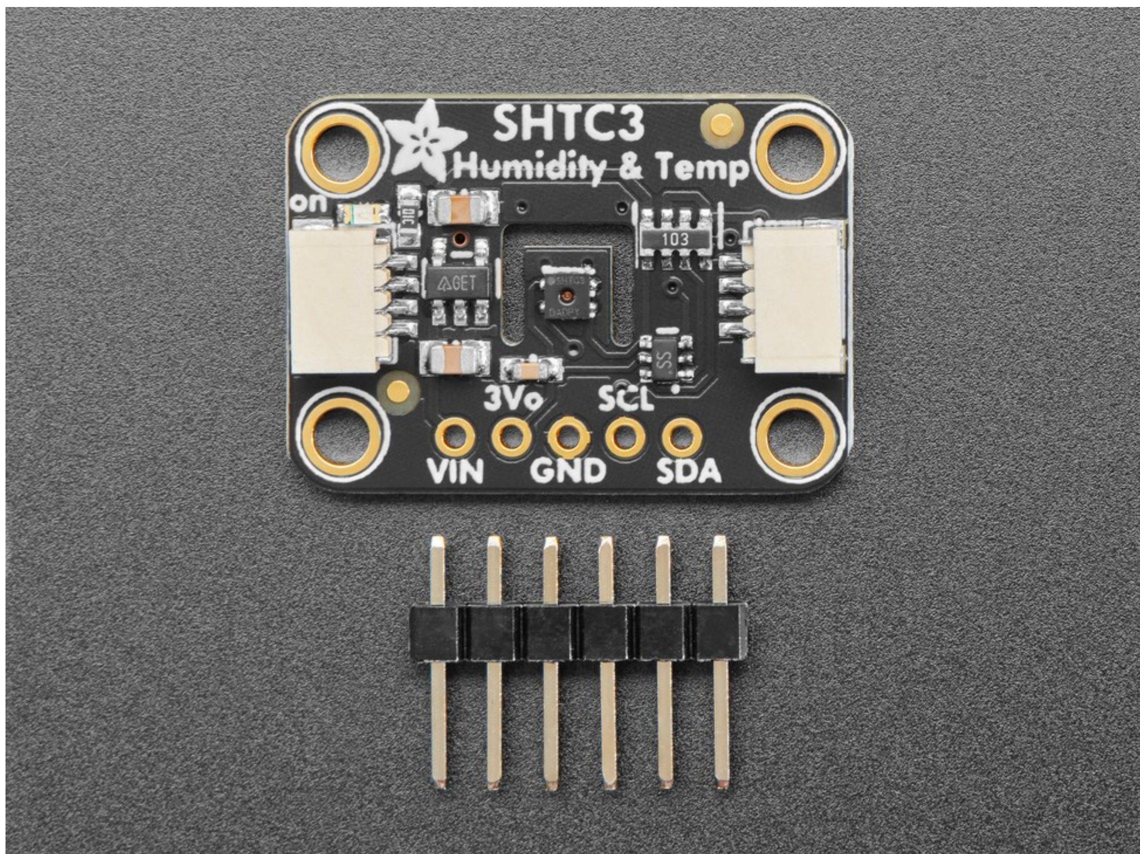

Figure 5. SHTC3 breakout board [5].

The SHTC3 sensor's I2c interface has a true I2C connection with only 2 wires and 2 more cables for power supply and ground, as opposed to some temperature and humidity sensors. The breakout has a regulator and level shift circuitry, allowing it to be 3V or 5V compliant. All microcontroller and microcomputers can be connected to the sensor.

2.2.4 BMP390 (Precision Barometric Pressure & Altimeter Sensor)

The BMP390 sensor is the next-generation of sensors from Bosch, and it is an upgrade to BMP280 and BMP388. It has a low altitude noise of 0,1 m and the same fast conversion time. This sensor can also be used in combination with I2C or SPI, as is the case for earlier BMP280 sensors. Figure 6 below demonstrates the breakout development board for BMP390 used in the project. [6.]

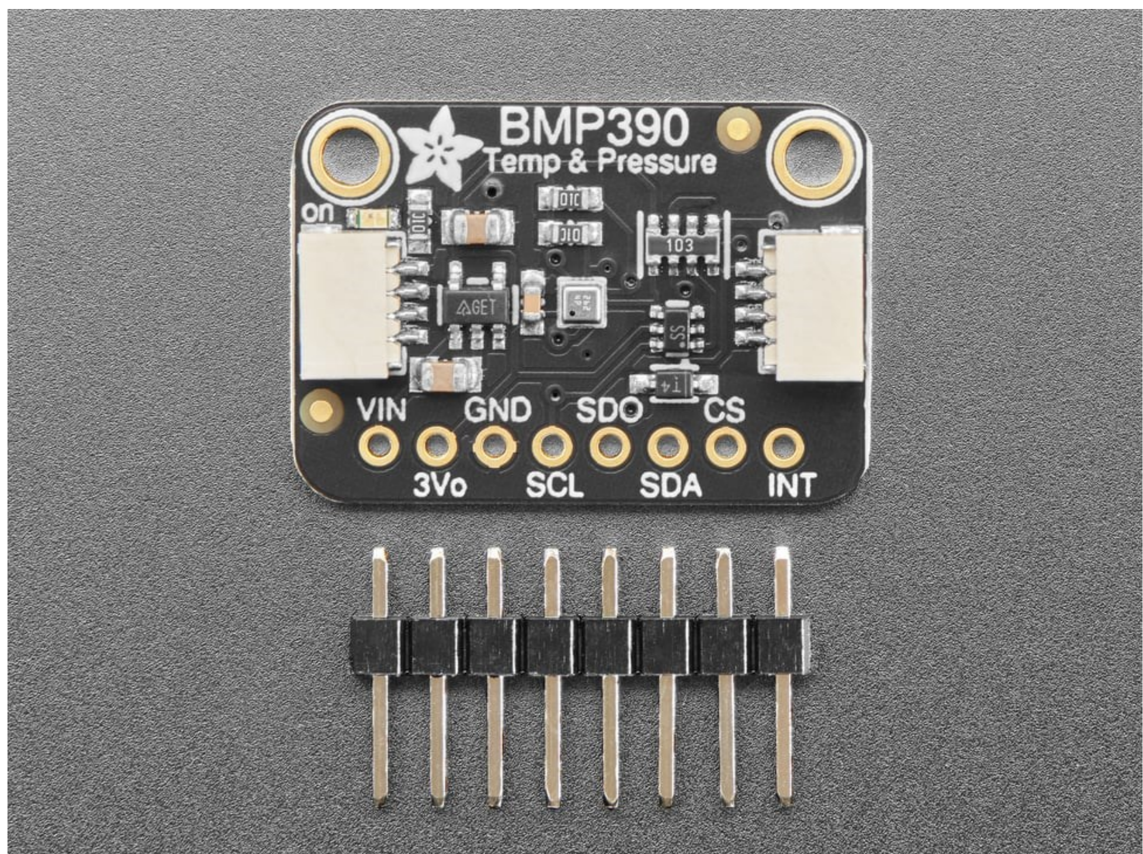

Figure 6. BMP390 breakout board [6].

This sensor has an approximate 3Pascal accuracy, which corresponds to about 0.25m of altitude. For the purpose of keeping altitude stable, those sensors should be used for drones and quadcopters, for wearables or in any project in which height-above-sea-level should be tracked. [6.]

2.2.5 VEML7700 (Lux Sensor)

For brighter ambient light, the majority of lights sensors only give you a number. Calculation of lux, a SI unit for light, is made easier by the VEML7700. The light sensor exhibits a dynamic range of 16 bits, enabling it to detect ambient light levels ranging from complete darkness (0 lux) to approximately 120,000 lux. It boasts a resolution as fine as 0.0036 lux per count (lx/ct) and features adjustable gain and integration times through software. Operating as an I2C sensor, it is compatible with a power supply in the range of 3.3 to 5 volts of direct current (DC). Figure 7 below demonstrates the breakout development board for VEML7700 which was used in the project. [7.]

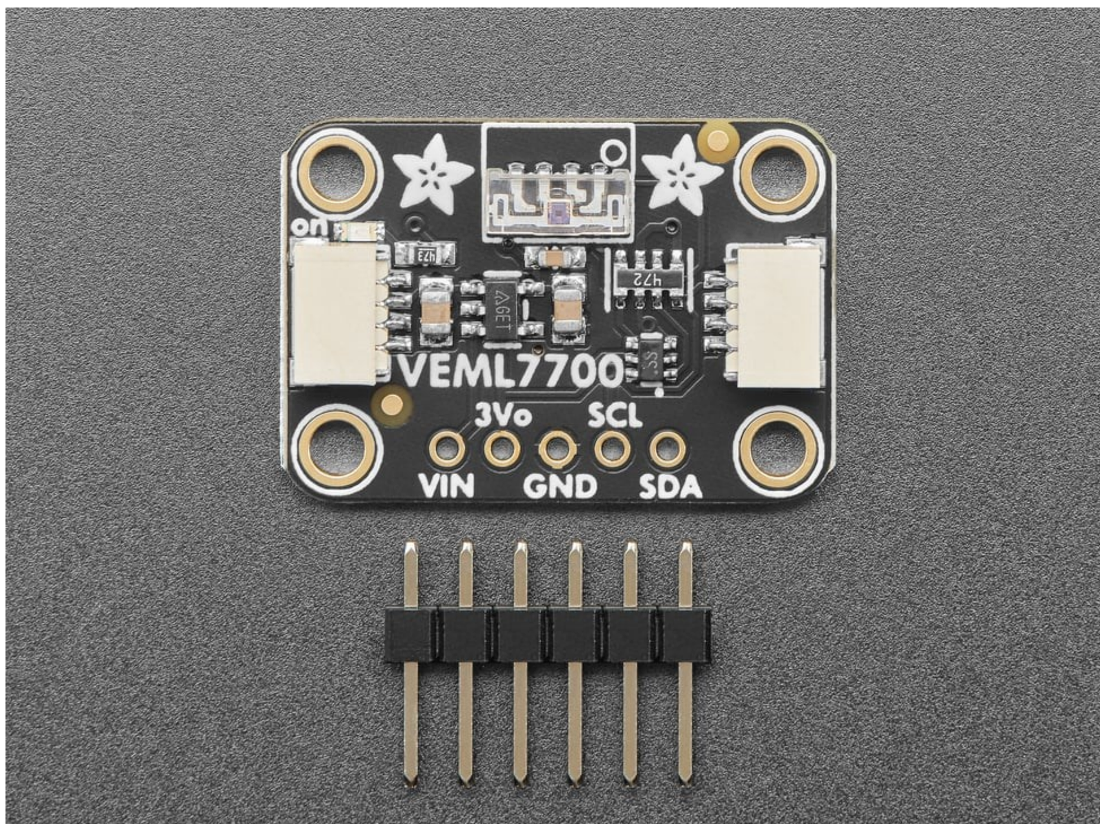

Figure 7. VEML7700 breakout board [7].

2.2.6 SGP30 (Sensor for Air Quality)

The SGP30 gas sensor is a high-quality air sensor made by Sensirion. It's designed to provide accurate air quality information, offering I2C interfacing and calibrated output signals. The sensor uses a set of metal oxide sensors on a single chip to enhance precision, typically with a 15% accuracy within measured values. It is a gas analyzer for detecting VOCs as well as H2 and intended to measure air quality in the home. The sensor will give TVOCs (Total Volatile Organic Compounds) value and an eCO2 over the I2C signal if you connect your microcontroller with proper code. Figure 8 below illustrates the breakout development board for SGP30 used in the project. [8.]

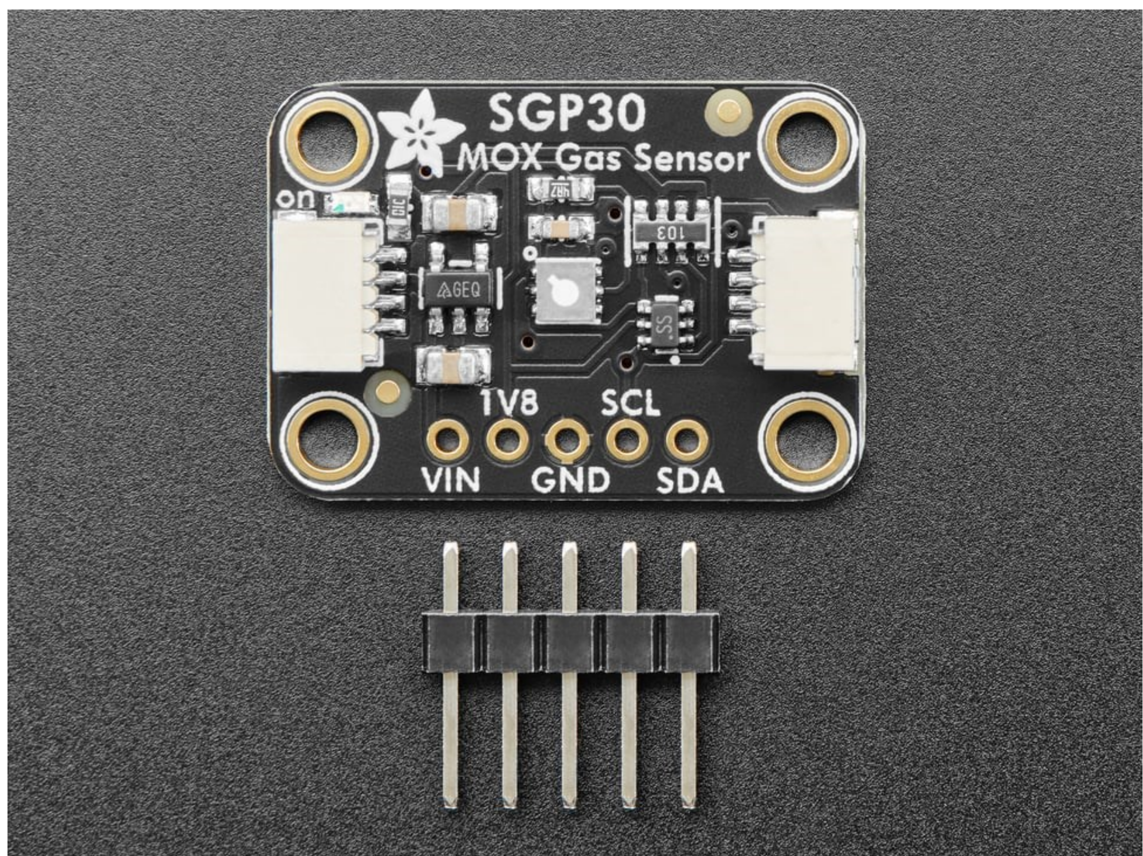

Figure 8. SGP30 breakout board [8].

This SGP30 sensor will measure eCO2 (equivalent calculated carbon-dioxide) concentration within a range of 400 to 60,000 parts per million (ppm), and TVOC concentration within a range of 0 to 60,000 parts per billion (ppb). [8.]

2.2.7 Minor Components

Other minor components include Raspberry Pi 4 power supply, a Raspberry Pi 4 case, a Micro SD card, a Micro SD card reader, a Micro USB cable for esp32, a STEMMA QT / Qwiic JST SH 4-pin cable with premium female sockets, a STEMMA QT / Qwiic JST SH 4-pin cable, a Jumper cable, double sided tape, a power bank, a water proof box, and PVC tape.

2.3 Software Selection

2.3.1 Raspberry Pi OS

Raspberry Pi OS is a free operating system built on Debian, specifically designed for Raspberry Pi hardware. It's the recommended OS for everyday use on a Raspberry Pi. The system includes a wide range of pre-compiled software packages—over 35,000 of them presented in a user-friendly format for convenient installation on the Raspberry Pi. Ongoing development focuses on enhancing the stability and performance of numerous Debian packages when running on the Raspberry Pi. [9.]

2.3.2 AWS IoT Core

AWS IoT Core is a robust and secure platform that empowers businesses to connect and manage a multitude of IoT devices seamlessly. It enables the easy exchange of data between devices and the cloud, facilitating real-time insights and actions. With its scalability, reliability, and extensive ecosystem of services, AWS IoT Core simplifies IoT development, making it an essential tool for harnessing the full potential of the Internet of Things. [10.]

2.3.3 Python3

Python stands out as an extensively utilized high-level programming language with broad applicability. Conceived by Guido van Rossum in 1991 and

subsequently enhanced by the Python Software Foundation, its core design principle emphasizes the readability of code. The language's syntax facilitates concise expression of programming concepts, enabling developers to convey ideas with brevity. Python, characterized by its efficiency in rapid development and seamless system integration, proves to be a versatile programming tool. [11.]

2.3.4 Python3 Libraries

Multiple Python libraries were installed to receive data from the AWS cloud, store it in the Raspberry Pi’s memory, analyze it and then finally send the processed data the Blynk IoT platform. These libraries were the following:

- AWS IoT Device SDK v2 for Python [12]
- Blynk Python Library [13]

2.3.5 PlatformIO

PlatformIO serves as a versatile, cross-platform, and cross-architecture tool, equipped with various frameworks tailored for both embedded systems engineers and software developers crafting applications for embedded products. It proves indispensable for professional embedded systems engineers engaged in multi-platform solution development. Its decentralized architecture facilitates a seamless integration process, enabling both new and experienced developers to swiftly create commercial-ready products, thereby minimizing the overall time-to-market. [14.]

2.3.6 PlatformIO Libraries

Different libraries were used to get data from the sensor and push the acquired data to the AWS cloud from ESP32. These libraries were the following:

- arduino-mqtt [15]
- ArduinoJson [16]

- Adafruit_VEML7700 [17]
- Adafruit-SGP30-PCB [18]
- Adafruit_BMP3XX [19]
- SparkFun_SHTC3_Arduino_Library [20]

3 System Implementation

To develop the field devices, a flowchart was designed as illustrated in Figure 9. The overall functionality of the system is based on the decision process presented in the flowchart. The system turns on. ESP32 will then try to connect to the Wi-Fi. If the Wi-Fi connection is not made, then the ESP32 will restart until the Wi-Fi connection is made. Once the Wi-Fi connection is made, then the system will establish connection to the AWS cloud. Upon successful cloud connection, the system moves towards an infinite loop of gathering data and sending the data to the AWS cloud.

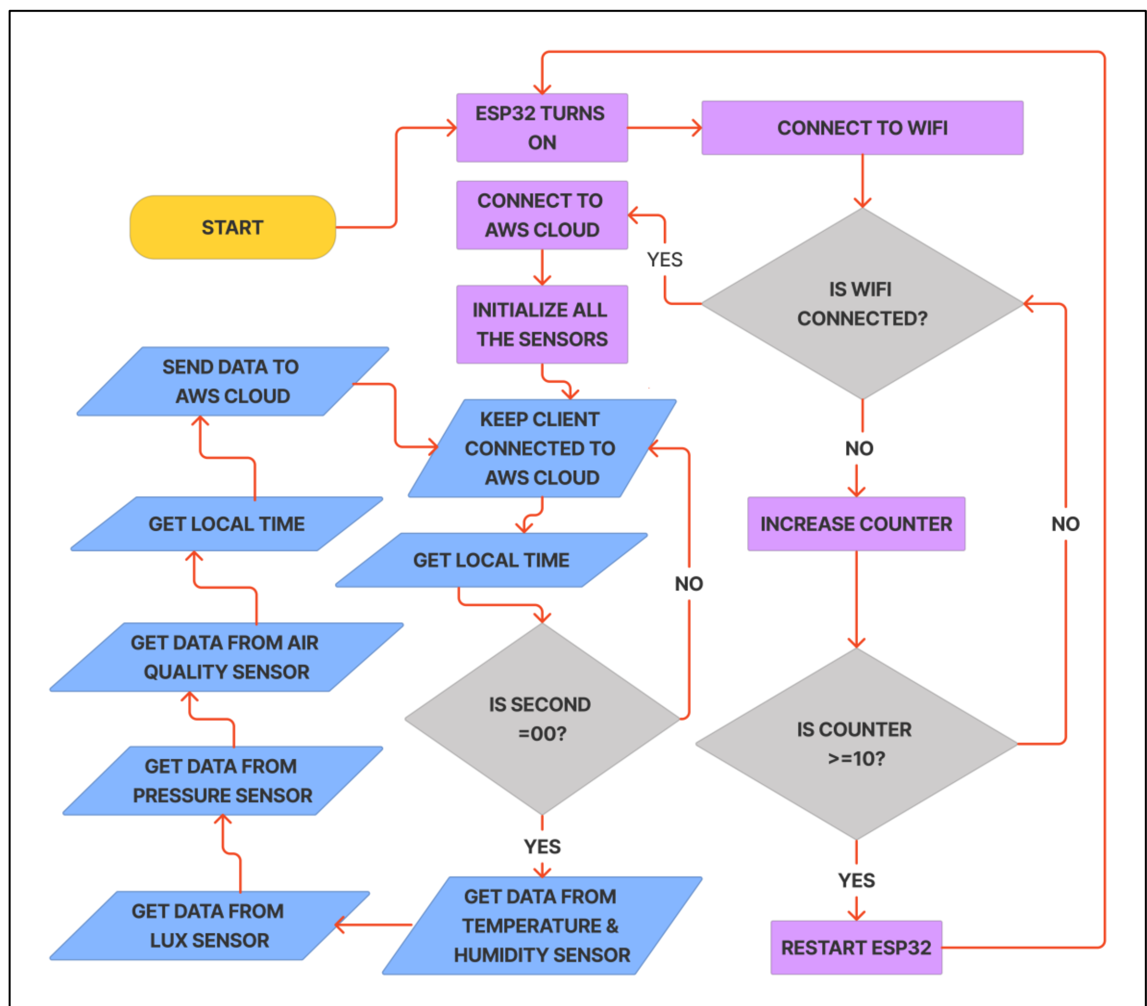

Figure 9. Flowchart for ESP32 based field devices.

The system gets local time from the internet using the NTP (network time

protocol) server. The system checks if the second value is zero second. When local time has zero seconds, the system collects data from the temperature and humidity sensor, the light intensity sensor, the atmospheric pressure sensor and the air quality sensor. Then the system grabs the whole timestamp from the NTP server. All these data are compiled into a single JSON (JavaScript object notation) object. This compiled JSON object is then published to the AWS cloud. The JSON object also has an ID key where the first ESP32 is ESP32-1 and the second one is ESP32-2. The system then waits for the second value to be zero again before repeating the process. In case of any failure, the same loop repeats over and over again.

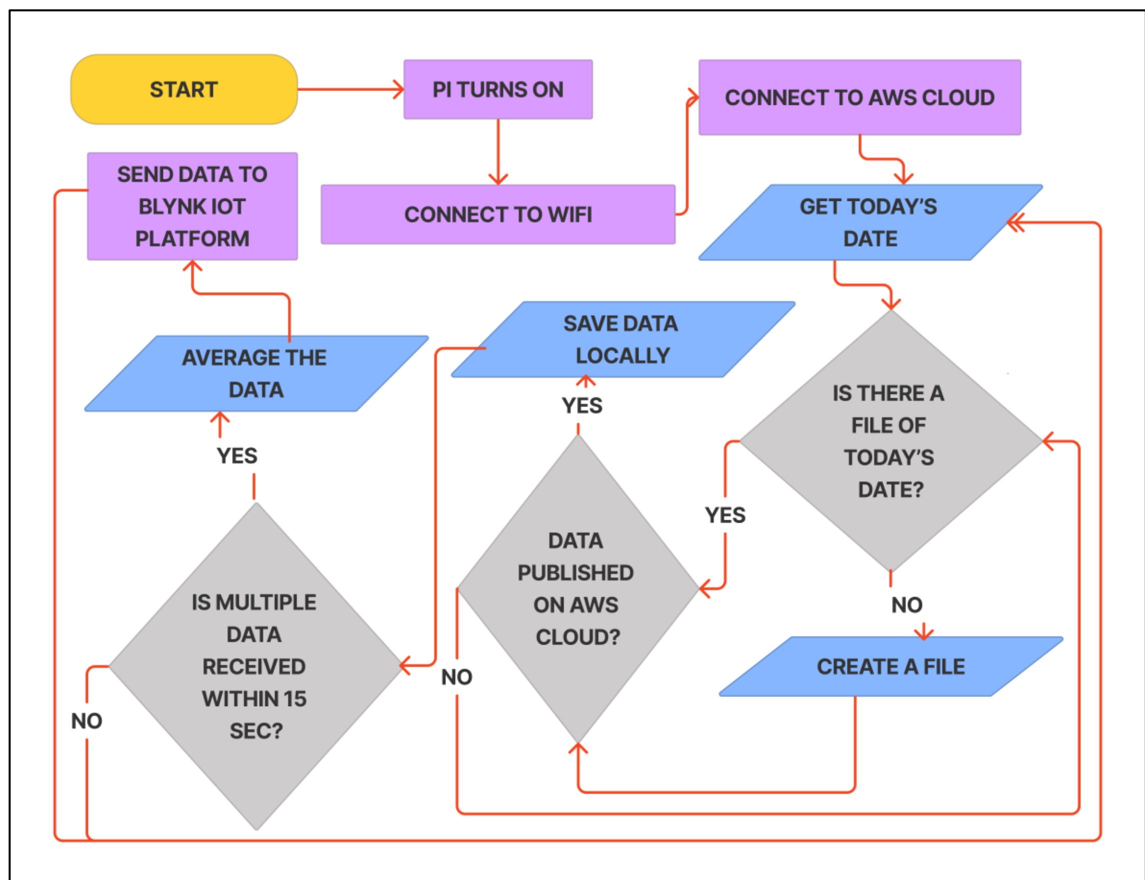

Figure 10. Flowchart for Raspberry Pi based master device.

In order to develop the master device, a similar flowchart was designed as illustrated in Figure 10. Since field devices are only supposed to send data to the AWS cloud, there has to be a master device, in this case a Raspberry Pi 4. The

Raspberry Pi would turn on then connection to the Wi-Fi. After that it would connect to the AWS cloud. The Raspberry Pi would subscribe to the topic that field devices are publishing to. The Raspberry Pi would check today's date and create a file named with today's date to store the data. If there already is a file then it would simply append the data to the file. Once the data are published from field devices, the data is received on the Raspberry Pi and stored locally. This data now is analyzed to see if similar data was received within 15 seconds. If yes, then all the data would be averaged and then published to the Blynk IoT platform for further viewing and analysis if needed.

Figure 11 below is the schematics that illustrates the hardware setup for the field devices that were used during the testing phase. The designed configuration was implemented and tested with hardware components. All the hardware represents the actual hardware in real settings.

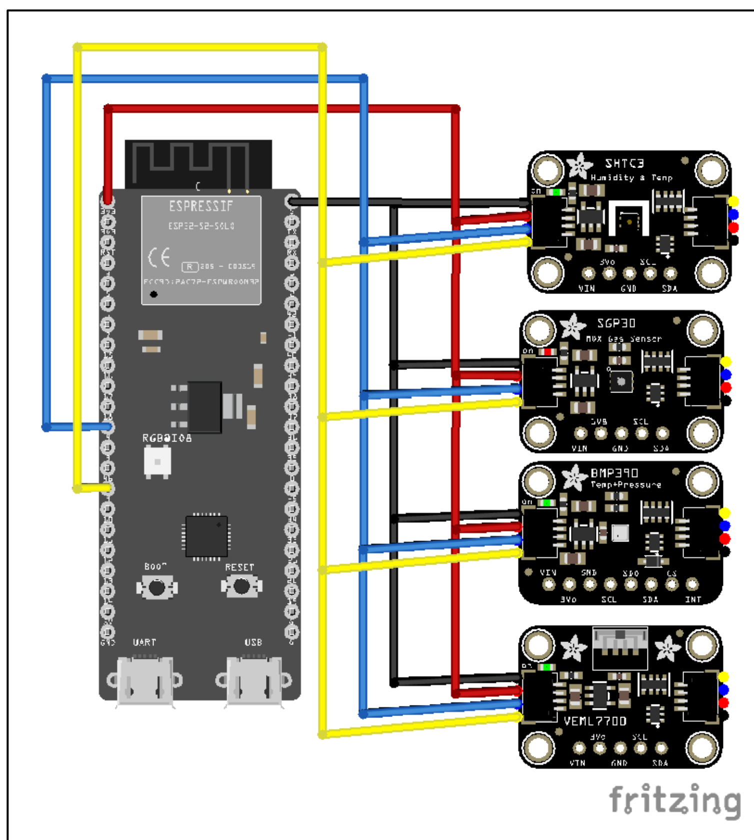

Figure 11. Schematic diagram for field device.

On the field devices side, VS (visual studio) Code was installed on a laptop. Then the Platform IO extension in the VS Code was installed so the sensors could directly configure, upload and monitor ESP32 from it. Then the temperature/humidity sensor with the ESP32 was connected, and the library was installed and tested. The temperature and humidity data were printed in the serial monitor. A similar method was applied for all of the sensors individually and the data were being printed on the serial console. Once all the sensors were tested, all the hardware components were connected as illustrated in the schematic above. Then the codes were compiled together and data from all the sensors were printed on the console. The output can be seen in the screenshot below in Figure 12.

Figure 12. Screenshot displaying data from field device in serial console.

Since there were two sets of identical field devices, each hardware was tested and found to be good. Figure 13 illustrates how the sensors were connected to ESP32 and to the laptop for testing purposes.

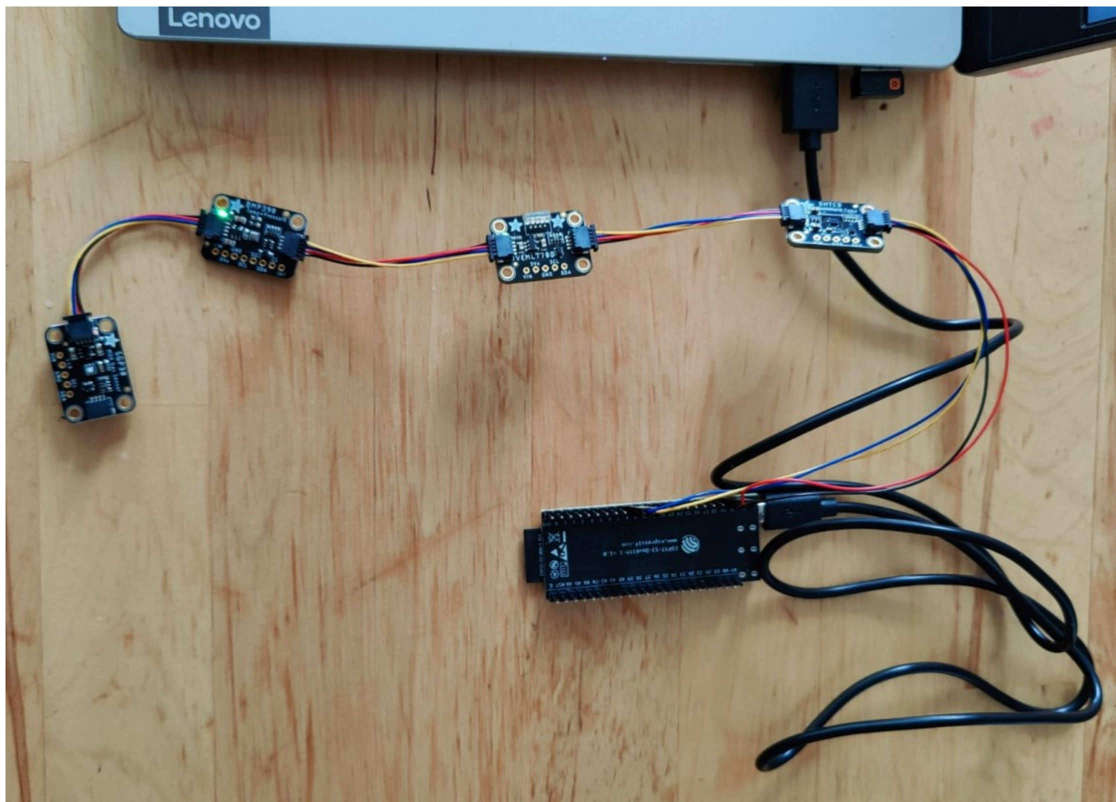

Figure 13. Field device connected to laptop for testing.

The hardware of the field devices was tested. Now it was time to send the collected data from individual field devices to the AWS cloud. A thing on the AWS IOT core was created and required certificates like Amazon root ca1, device certificate, private key, and public key were generated. These keys and certificates were downloaded on the host machine and saved in the same folder where the firmware of ESP32 was saved. Then the MQTT (message queuing telemetry transport) library was used along with these certificates to establish the connection between AWS and ESP32. For the testing purpose some random data was pushed to the server. Using the MQTT test client in AWS, the IOT core data were observed to be receiving. The output can be seen in the console as illustrated in Figure 14 below.

Figure 14. Screenshot of the serial console for data sending from field devices to the AWS cloud.

The code from the sensor's data acquisition, the AWS code and the MQTT code were all compiled together in a single code. A JSON object carried all the information that had to be sent to the AWS cloud. The data was observed in the AWS IoT core test client. Figure 15 below illustrates the same data being printed on the AWS test client console which was seen in Figure 15.

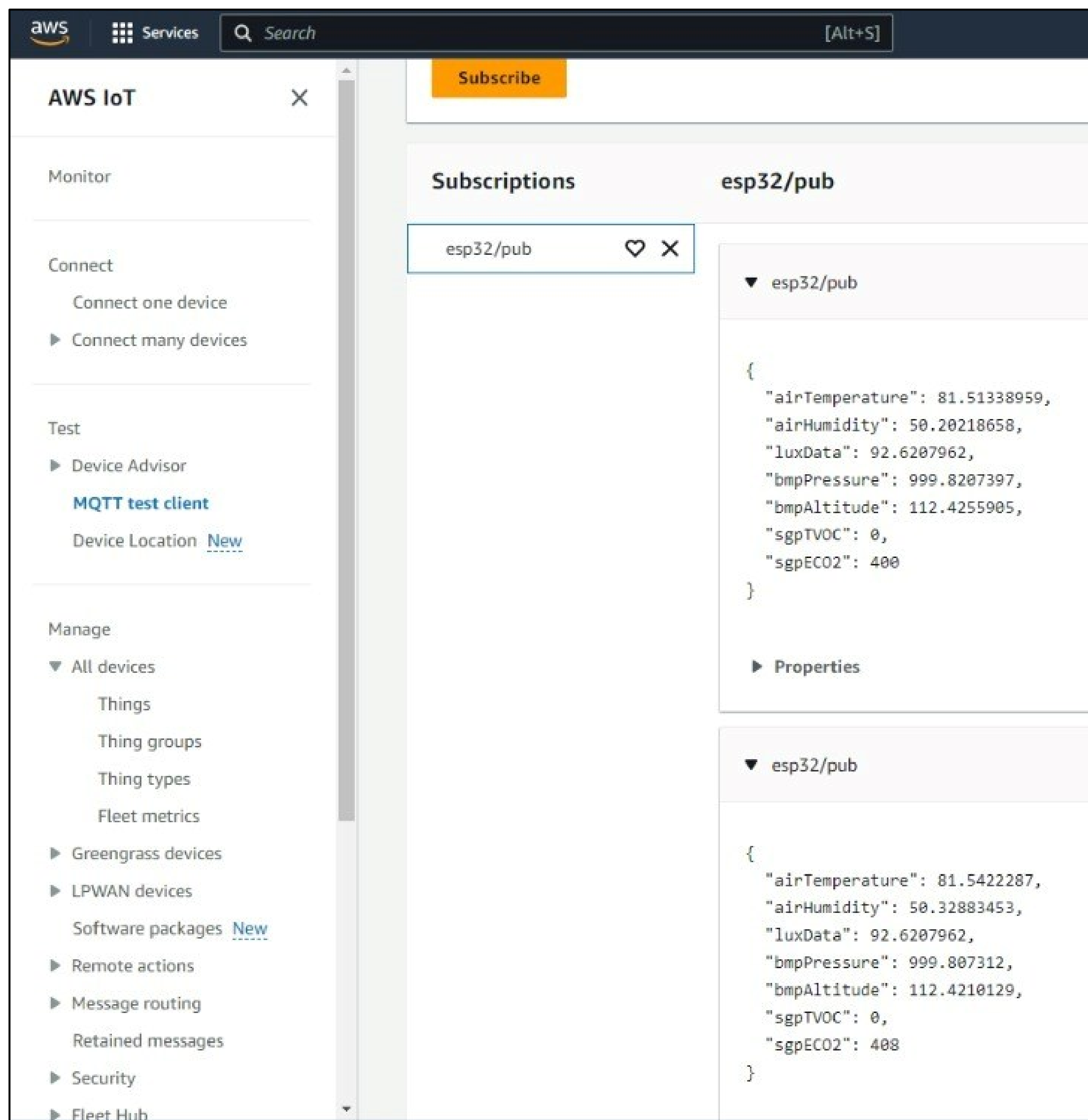

Figure 15. AWS cloud MQTT test client screenshot for data receiving.

The setup was sending data from both ESP32s in the following format.

{"id":"ESP32-1",
"timestamp":"Sun Aug 20 00:00:00 2023",
"airTemperature":24.05203247,
"airHumidity":48.71900558,
"luxData":1.382400036,
"bmpPressure":1019.403687,
"bmpAltitude":51.21237946,
"sgpTVOC":0,

"sgpECO2":400}

The first ESP32 uses this format and the second Esp32 also sends the data in a similar format. The only difference is that the ID is ESP32-2. The feature of getting current time is being received from the NTP server using the internet for both field devices. Both ESP32s check time parameters and when seconds are zero, then the data is sent to the AWS cloud. ESP32-1 publishes data to "esp32-1/pub" and ESP32-2 publishes data to "esp32-2/pub".

Another thing was created on the AWS IOT core for Raspberry PI and generated the required certificates like Amazon root ca1, device certificate, private key, and public key. Then awsiotsdk library was used along with these certificates to establish a connection between AWS and Raspberry Pi.

After this, Raspberry Pi subscribes to both of the topics for the data published from both ESP32s. Once the data are received, the data is appended into a text file and saved locally. Every day a new file with the date is created and the data are appended into that. Furthermore, when the data are received from both ESP32s, Raspberry Pi checks for the time difference between the two timestamps received from ESP32, and if they are in between 15 seconds, then it calculates the average of the values from the sensor and appends that to the same file. The data is appended as Figure 16 below shows.


```
{"id":"ESP32-1","timestamp":"Sun Aug 20 08:04:00 2023","airTemperature":23.59540939,"airHumidity":46.9
{"id":"ESP32-2","timestamp":"Sun Aug 20 08:04:00 2023","airTemperature":24.52734947,"airHumidity":46.5
{"id": "avgData", "timestamp": "2023-08-20 08:04:00", "airTemperature": 24.061379430000002, "airHumidi
{"id":"ESP32-1","timestamp":"Sun Aug 20 08:05:00 2023","airTemperature":23.67017746,"airHumidity":46.8
{"id":"ESP32-2","timestamp":"Sun Aug 20 08:05:00 2023","airTemperature":24.65018845,"airHumidity":46.6
{"id": "avgData", "timestamp": "2023-08-20 08:05:00", "airTemperature": 24.160182955000003, "airHumidi
```


Figure 16. Data appending in the text file locally in Raspberry Pi.

After that, the average data is published to the Blynk IoT cloud platform for further viewing. The Blynk platform was set up and the Blynk library was used to send data. The screenshot of the dashboard in Blynk is illustrated in Figure 17 below.

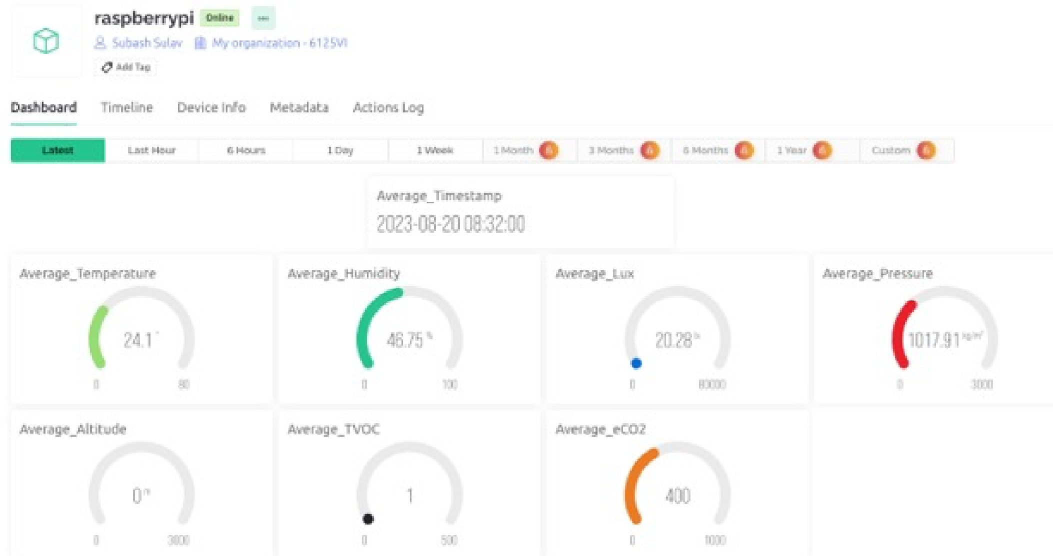

Figure 17. Blynk Console for data viewing and further analysis.

The Python script in Raspberry Pi is automated to run at boot using the pm2 library. At the moment both ESP32 are sending data to the AWS cloud, and Raspberry Pi is receiving data from the AWS cloud, saving the data locally, averaging out the data if they are published at the same time, and then finally sending them to the IoT platform for viewing purposes.

Once each test was done and the hardware and software were fully integrated into a full-fledged system, it was time to organize the field devices and the main devices into a PVC box to look organized. Then, the box could be placed outdoors for data collection and further testing.

The ESP32 sensor was placed inside of the PVC box and glands were used to redirect the wire from inside to outside as shown in Figure 18 below.

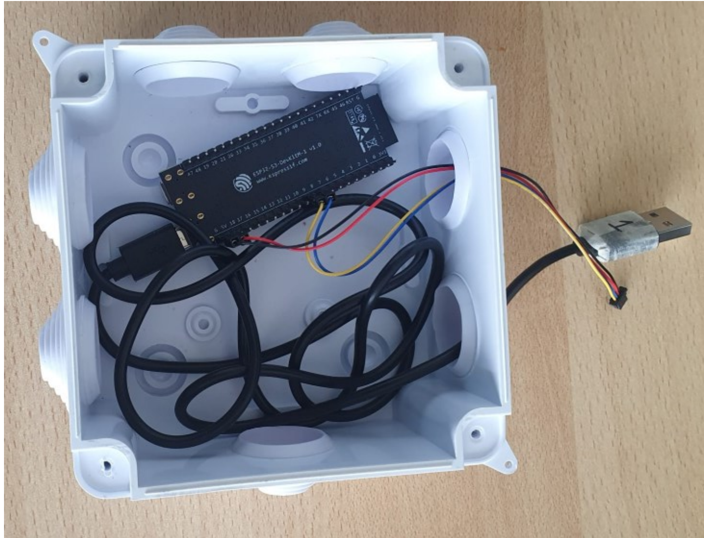

Figure 18. ESP32 inside PVC box.

The USB cable was to provide power supply to the ESP32 sensor and the sensors connected to it, and the qt cable was to connect the sensors in a daisy chain. The USB cable was connected to the power bank so the whole file device could be placed anywhere without the need of power supply from the wall as illustrated in Figure 19 below.

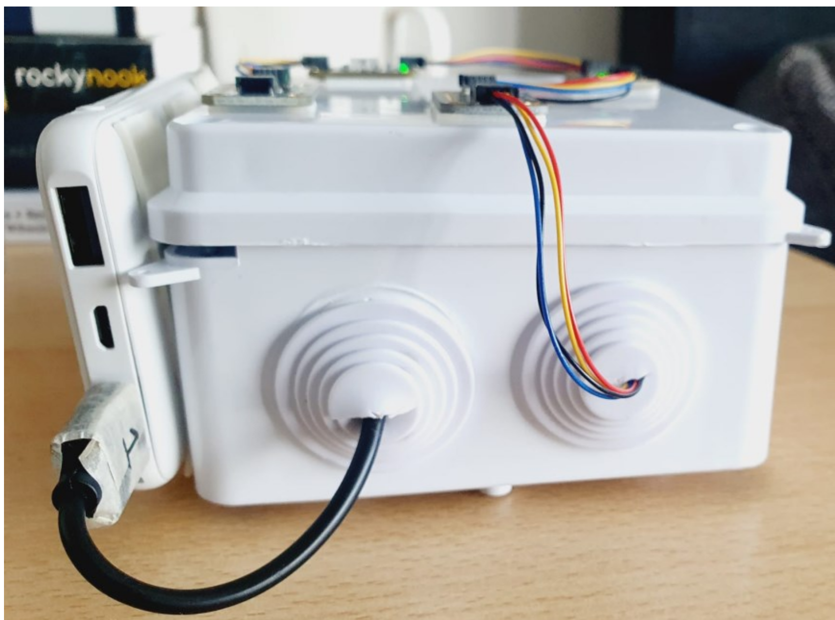

Figure 19. Wire coming out from the PVC box via glands.

The sensors were placed in the top cover of the PVC box. All four sensors were connected on the same I2C bus. With the help of the qt-to-qt cable, the sensors were daisy chained and sticked on the cover using double tape. The power bank was stuck on the side as shown in Figure 20 below.

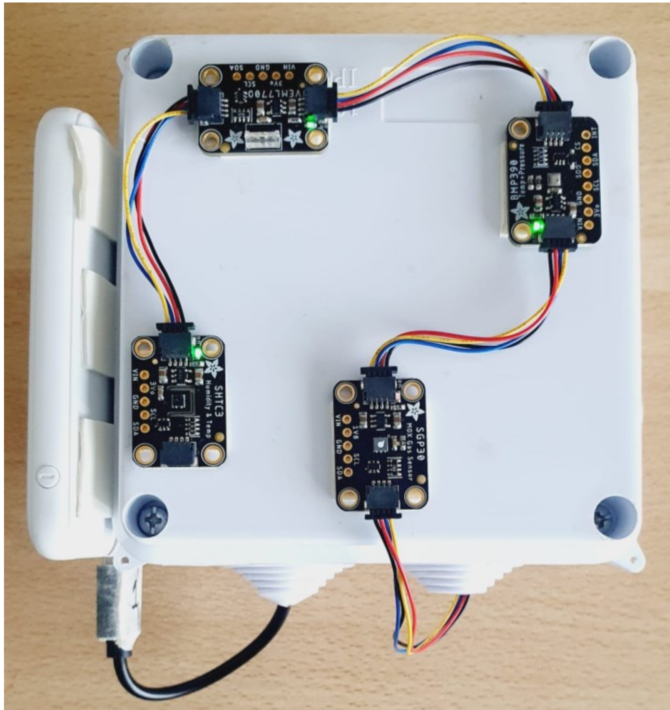

Figure 20. Demonstration of sensors on top of PVC box.

The same identical setup was also created for the second field device, as demonstrated in the Figure 21 below.

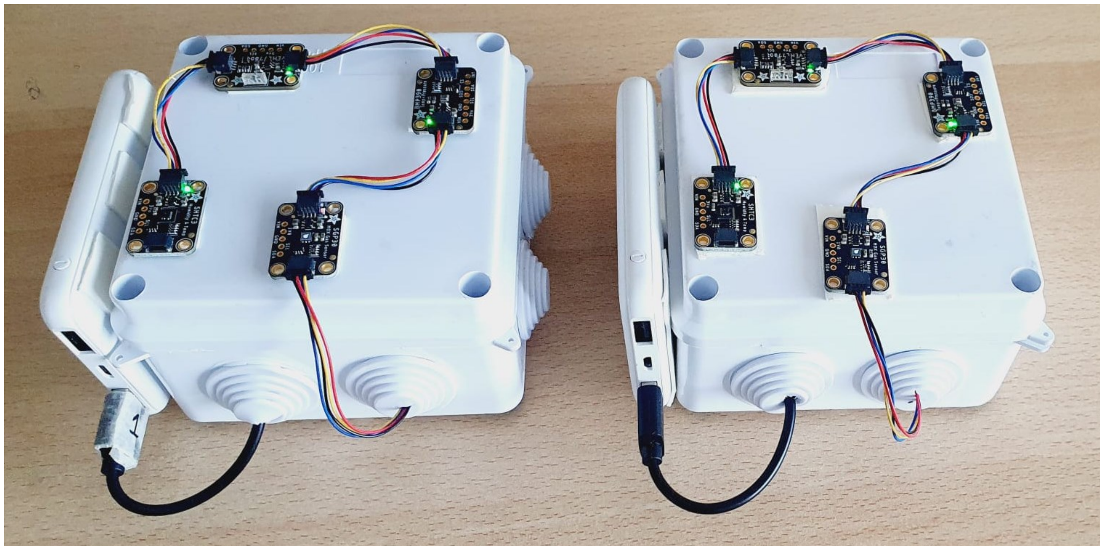

Figure 21. Illustration of both identical field devices.

After this, Raspberry Pi was also placed inside the PVC box, so there was no need to worry about the harsh outdoor conditions. This can be seen in the Figure 22 below.

Figure 22. Raspberry Pi inside PVC box.

Since Raspberry Pi only needed a power supply and the rest was all happening via Wi-Fi, only the power cable was brought outside using the gland. Since the master device does not have to be a specific location, it was connected to the AC (alternating current) wall socket for power supply.

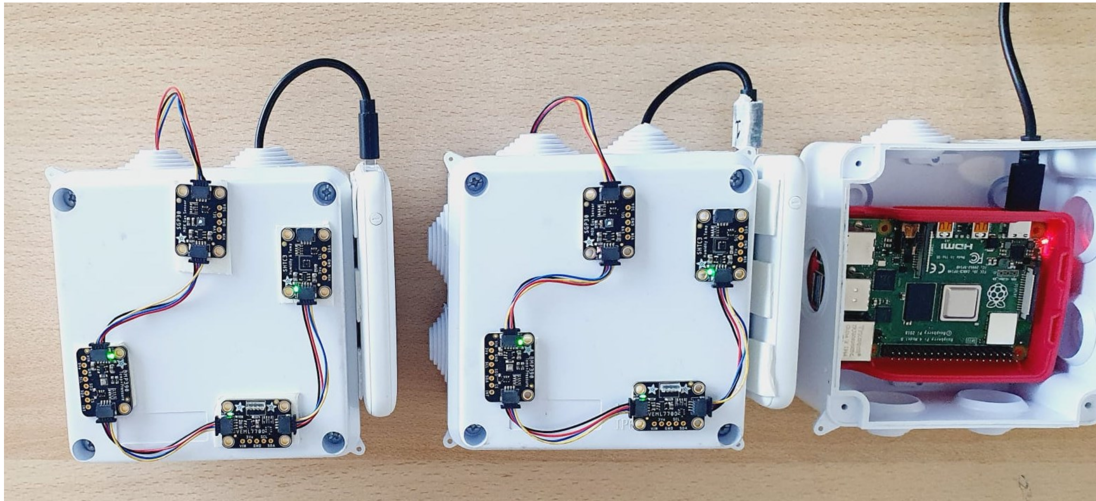

Figure 23. Demonstration of all field devices and master device together.

Figure 23 above shows two identical field devices and a master device together in a single picture. After this, the field devices were placed in two different locations and the master device was connected to the power socket and left to be tested for further analysis.

4 Results

When the initial system was implemented, it was found out that there were issues with the SHTC3 library. The sensors would work individually but when they were all connected in a daisy chain, the temperature and humidity sensor SHTC3 would stop sending data to ESP32. The used libraries were Adafruit's libraries. After this issue, the SHTC3 library from Sparkfun was used and the issue was solved. For some reason the altitude also had a negative value. This was because the altitude was derived from the atmospheric pressure. Since the altitude can never be negative, the absolute value of the data was taken so the altitude always has/had a positive value no matter what.

Once these issues were fixed, the system was run for a whole week and there were a couple of times where the data was not being sent from either of the field devices. A piece of code was instructing / instructed ESP32 to run only when a/the serial monitor was available. As a result, when ESP32 would reboot, then it would get stuck at that point and would not send data. The code was modified to run without checking the availability of the serial monitor. Even if the serial monitor was not available, the code would run normally. This way the issue was also fixed. After fixing this issue, everything was solved and the whole system worked flawlessly.

Hence, only the data received after all the issues were fixed was taken into consideration to draw conclusions. The data obtained on August 20th, 2023 was taken into consideration and the data analyzed are presented below in Figure 25-Figure 31. Figure 24 demonstrates the data being logged in the text file in Raspberry Pi.

{"id": "avgData", "timestamp": "2023-08-20 06:35:00", "airTemperature": 24.28435135, "airHumidity": 47.090867994999996, "
{"id":"ESP32-1","timestamp":"Sun Aug 20 06:36:00 2023","airTemperature":23.73159409,"airHumidity":47.19005203,"luxData":5.
{"id":"ESP32-2","timestamp":"Sun Aug 20 06:36:00 2023","airTemperature":24.65018845,"airHumidity":47.00083923,"luxData":3.
{"id": "avgData", "timestamp": "2023-08-20 06:36:00", "airTemperature": 24.19089127, "airHumidity": 47.09544563, "luxData"
{"id":"ESP32-1","timestamp":"Sun Aug 20 06:37:00 2023","airTemperature":23.73159409,"airHumidity":47.42961884,"luxData":5.
{"id":"ESP32-2","timestamp":"Sun Aug 20 06:37:00 2023","airTemperature":24.65018845,"airHumidity":46.8589325,"luxData":3.
{"id": "avgData", "timestamp": "2023-08-20 06:37:00", "airTemperature": 24.19089127, "airHumidity": 47.14427567, "luxData"
{"id":"ESP32-1","timestamp":"Sun Aug 20 06:38:00 2023","airTemperature":23.79301071,"airHumidity":47.30449295,"luxData":5.
{"id":"ESP32-2","timestamp":"Sun Aug 20 06:38:00 2023","airTemperature":24.58877182,"airHumidity":46.83146667,"luxData":3.
{"id": "avgData", "timestamp": "2023-08-20 06:38:00", "airTemperature": 24.190891265, "airHumidity": 47.06797981, "luxData
{"id":"ESP32-2","timestamp":"Sun Aug 20 06:39:00 2023","airTemperature":24.65819931,"airHumidity":46.79484558,"luxData":4.
{"id":"ESP32-1","timestamp":"Sun Aug 20 06:39:00 2023","airTemperature":23.79301071,"airHumidity":47.29076004,"luxData":5.
{"id": "avgData", "timestamp": "2023-08-20 06:39:00", "airTemperature": 24.225605010000002, "airHumidity": 47.04280281, "
{"id":"ESP32-1","timestamp":"Sun Aug 20 06:40:00 2023","airTemperature":23.79301071,"airHumidity":47.36400604,"luxData":5.
{"id":"ESP32-2","timestamp":"Sun Aug 20 06:40:00 2023","airTemperature":24.51933861,"airHumidity":46.88182068,"luxData":4.
{"id": "avgData", "timestamp": "2023-08-20 06:40:00", "airTemperature": 24.156174659999998, "airHumidity": 47.12291336, "
{"id":"ESP32-2","timestamp":"Sun Aug 20 06:41:00 2023","airTemperature":24.54871178,"airHumidity":46.86808777,"luxData":4.
{"id":"ESP32-1","timestamp":"Sun Aug 20 06:41:00 2023","airTemperature":23.73159409,"airHumidity":47.25108719,"luxData":6.
{"id": "avgData", "timestamp": "2023-08-20 06:41:00", "airTemperature": 24.140152935000003, "airHumidity": 47.05958748000

Figure 24. Data compiled from Raspberry Pi after processing.

It can be clearly seen that at every minute at zero second time, the data are being sent from the field devices to the main device. The main device is also receiving data from both field devices and is able to calculate the average of the data and also append the data into the text file. The ID section defines the data coming in from the field devices or averaged on the Pi on itself. Based on Figure 24, it can be clearly observed that after every two field device data there is averaged data with the avgData tag on the ID.

Based on the file saved on August 20th 2023, the following analysis were drawn.

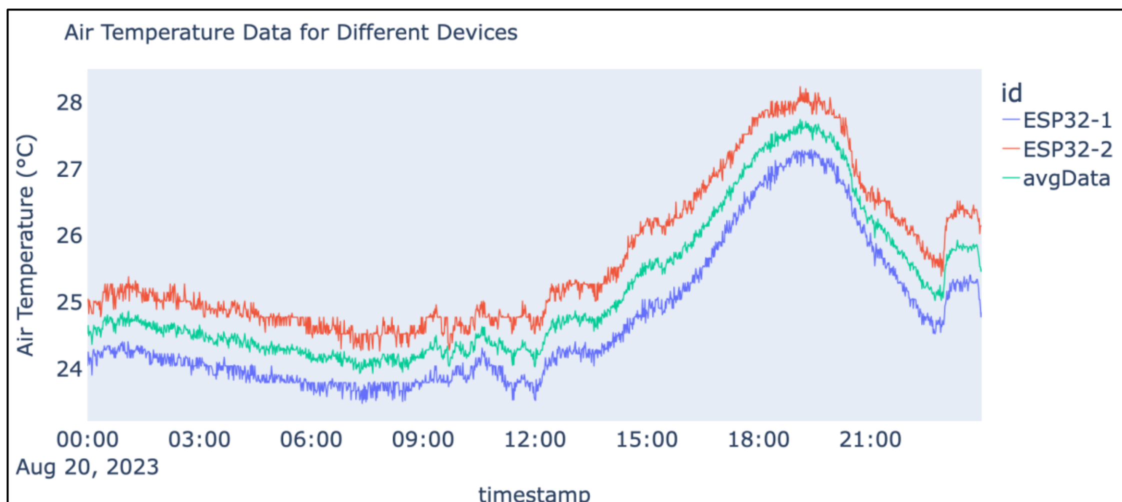

Figure 25. Air temperature versus timestamp graph plots for all three datasets.

Based on Figure 25 above, it can be clearly seen that both the field devices are sending data periodically and the master device is also receiving the data and averaging it. The red line represents the data sent by the second field device, the blue line represents the data sent by the first field device and finally the green line represents the data averaged by the main device. The green line lies in between the red and blue line which illustrates that the field devices can be used to sample data and the main device can draw average result based on these field devices.

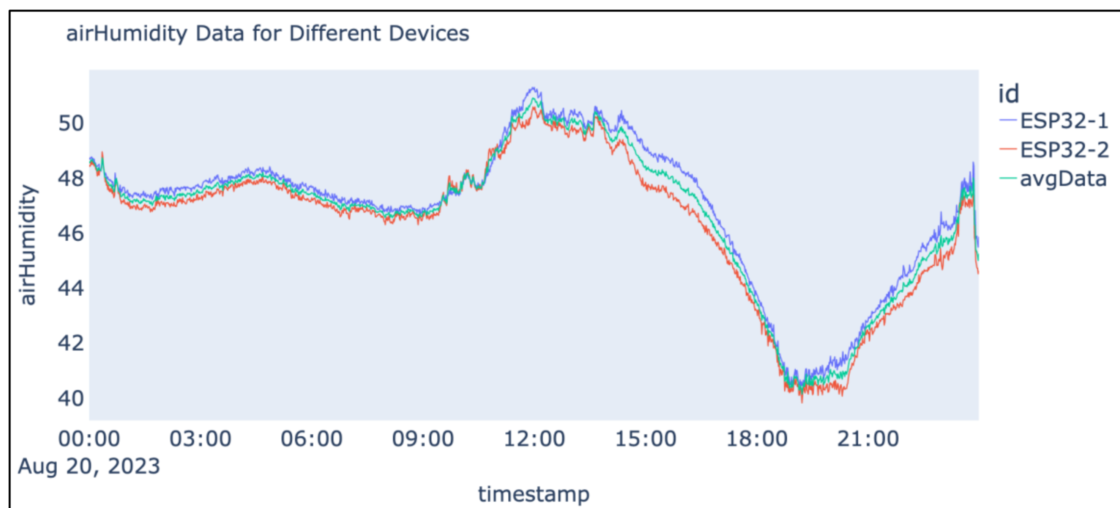

Figure 26. Air humidity versus timestamp graph plots for all three datasets.

A similar result can be seen in Figure 26. The green line lies in between the red and the blue line throughout the plot. The graph clearly shows that the humidity data from both of the field devices are being sent to the AWS cloud and being received from the AWS cloud by the main device.

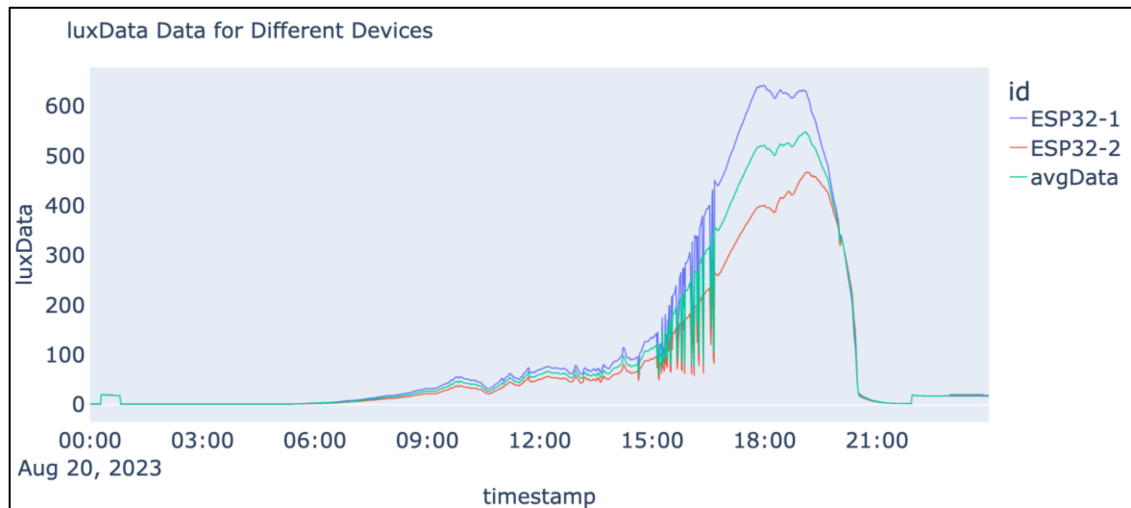

Figure 27. Light intensity versus timestamp graph plots for all three datasets.

Figure 27 above demonstrates the lines of light intensity for all three data sets. It seems like the light intensity inside the room went up to 600 lux. It can be clearly seen that the green line is following the pattern of the blue and red lines. The averaged data is clearly being drawn from the field devices and hence seems pretty solid and tangible.

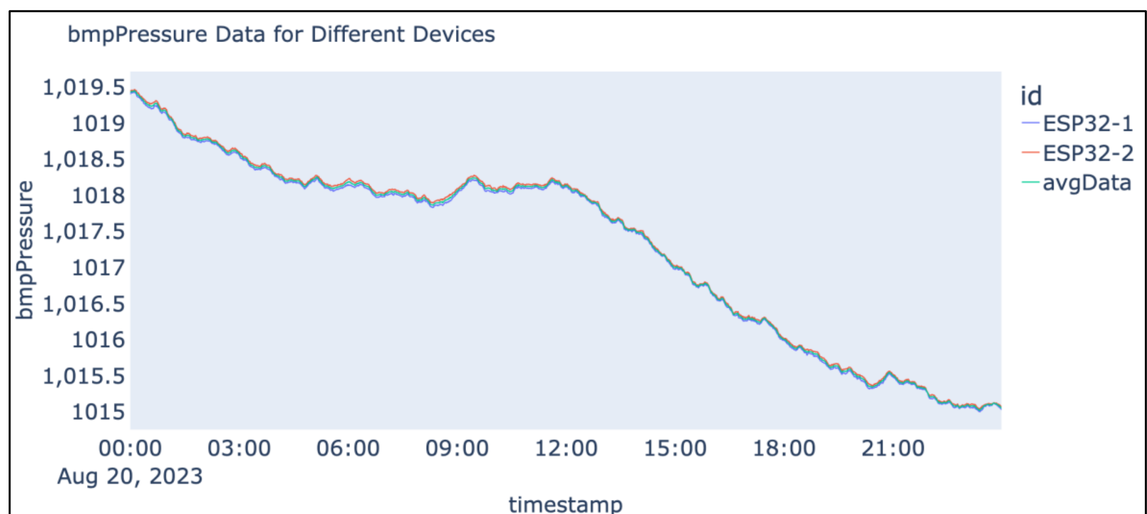

Figure 28. Barometric pressure versus timestamp graph plots for all three datasets.

The barometric pressure data sent from the two field devices and the average data obtained from the main device can be plotted as in Figure 28 above. Since

the pressure data does not change that much within a town or city, it can be clearly seen that all three colored lines are overlapping into one another. However, it can be concluded that the data are being received and averaged altogether to draw a result.

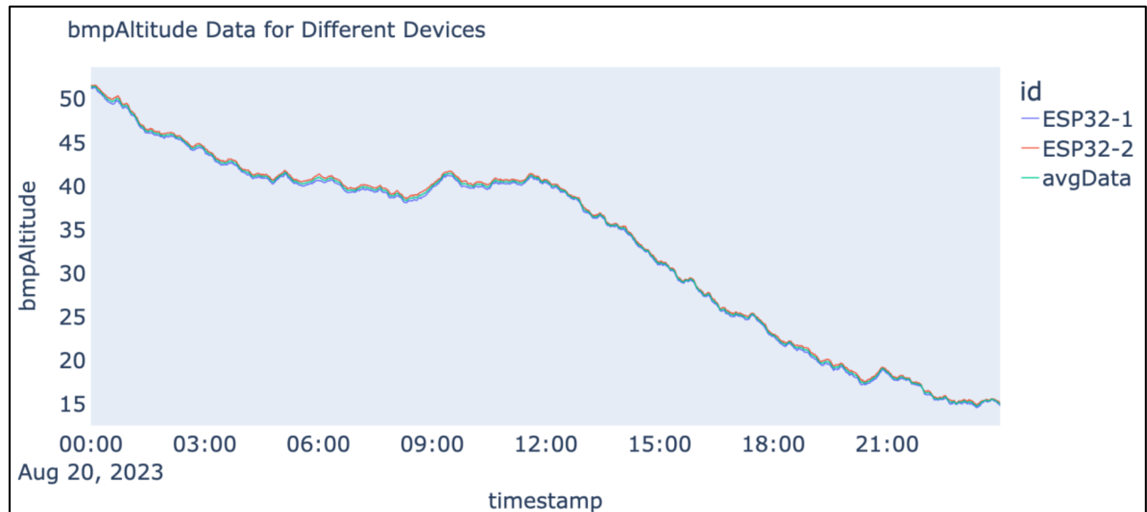

Figure 29. Altitude versus timestamp graph plots for all three datasets.

Figure 29 depicts the plots between the altitude and time of both field devices and the master device's averaged data. The barometric pressure's data follows exactly the same pattern as altitude's data. The altitude data is derived from the pressure data internally within the sensor. Both graphs look identical.

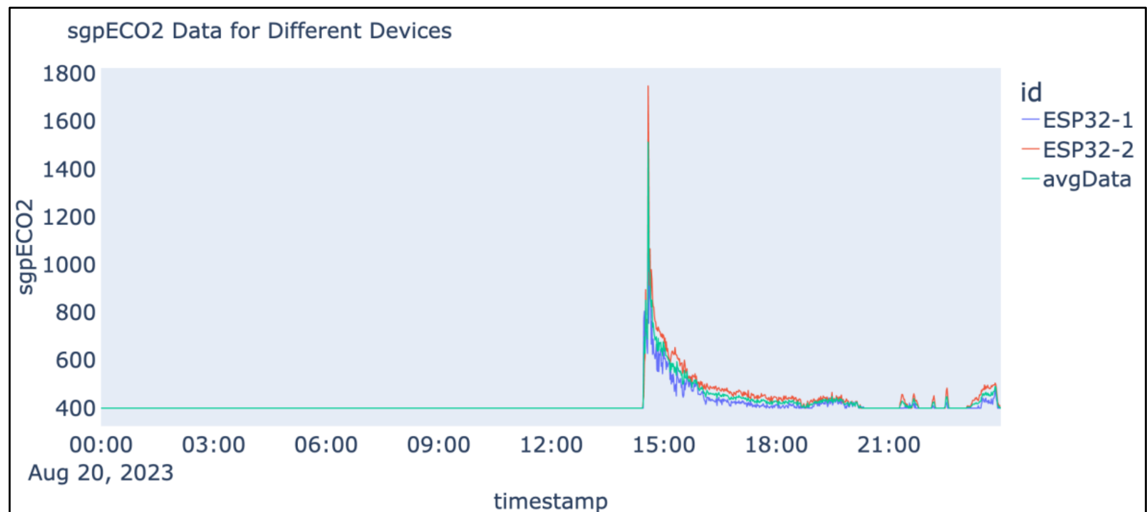

Figure 30. eCO2 versus timestamp graph plots for all three datasets.

Figure 30 above demonstrates the graph of eCO2 versus timestamp for all three data sets. The data seems to be pretty consistent around 400 for eCO2 until 3 pm and after that the data went high up to 1700-1800 and dropped down. In general, all three-color lines can be traced.

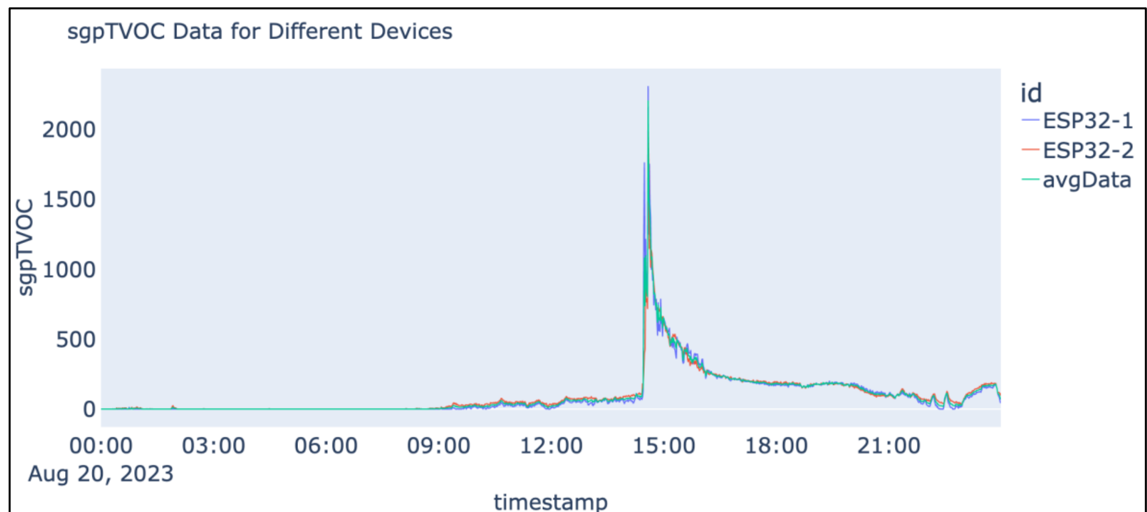

Figure 31. TVOC versus timestamp graph plots for all three datasets.

Figure 31 illustrates TVOC versus timestamp, which also follow a similar pattern to eCO2 as illustrated in Figure 30. It can be concluded that throughout the plots, the green line which is the plot of averaged data is always in the middle of data coming from two field devices.

5 Conclusions

The main objectives of this project were to develop two sets of identical ESP32 based field devices which would be able to get data from different environmental sensors and to publish the data to cloud services such as AWS. Then the project aimed to develop a main control unit based on Raspberry Pi which would be able to subscribe the published data from the field devices and store them locally as well as publish them on the IoT cloud platform for viewing as well as further analysis. After both of the devices were developed, the goal was to test the designed system placing the first ESP32 based field device in a different location from the second ESP32 based field device, which would then send data to the main device in a totally different location.

In order to develop the proposed solution, systems were designed for both field devices and the master device. Proper hardware and software were selected for both of the systems.

Then all the systems were implemented based on the flowchart developed for the field devices and the master device. All sensors were tested separately. Once all the data were verified, the actual construction of the field devices and the master device was done. The field devices and the master device were programmed in such a way that the required objective was met properly and left for testing in real settings.

Though multiple issues were encountered during the development work, such as issues with the library, SHTC3 not sending data when connected with other I2C sensors, and the altitude sensor giving negative values, all these issues were resolved and addressed properly. Eventually the whole system was functional and collected and stored data in the main device. Hence, a comprehensive analysis was conducted using data to draw results.

In conclusion, two sets of identical ESP32 based field devices were developed in this final year project. The devices were able to get data from different environmental sensors and publish the data to cloud services such as AWS. In

addition, a main control unit was developed based on Raspberry Pi, which would be able to subscribe the published data from the field devices explained above, store them locally and publish them in the IoT cloud platform for viewing as well as further analysis.

The designed system was also tested by placing the first ESP32 based field device in a different location from the second ESP32 based field device which would then send data to the main device in a totally different location. The data then confirmed that a cheap alternative sensing and monitoring system could be tested out through real life practices and that it has a lot of potential.

It can be concluded that the cheap alternative sensing and monitoring system that was experimented with in this project with a master-slave architecture, where the master can collect data from several slaves or field devices, can be practiced in a real-life scenario. Big cities, big greenhouses and large industries where several data setpoints are needed in order to summarize the environmental data of the whole can also implement this architecture. These smaller, inexpensive, affordable systems can be placed totally isolated from the main master devices while still sending data to the main master devices in real time. The failure of a slave device had no impact on the other field devices or even the master device. The developed solution was rather robust and turned out to be feasible for the real use case.

Furthermore, the architecture could be tested even more with many field devices in the near future. The whole solution could be tested in an industrial grade application and examined to see if this inexpensive system would perform in the same way even in harsh industrial conditions. The results could be compared with the existing, expensive industrial systems. One could also conduct research on designing the exterior housing for the system for better protection and improved sensor placement to collect data more effectively.

References

1 Raspberry Pi 4 Model B Specifications – Raspberry Pi. n.d. https://www.raspberrypi.com/products/raspberry-pi-4-modelb/specifications/ (Accessed November 7, 2023).

2 Meet Raspberry Pi | Getting Started with Raspberry Pi | Coding Projects for Kids and Teens. n.d. https://projects.raspberrypi.org/en/projects/raspberry-pi-getting-started/2 (Accessed November 7, 2023).

3 ESP32-S3 Wi-Fi & Bluetooth 5 (LE) MCU | Espressif Systems. n.d. https://www.espressif.com/en/products/socs/esp32-s3 (Accessed November 7, 2023).

4 ESP32-S3-DevKitM-1 - ESP32-S3 - — ESP-IDF Programming Guide Latest Documentation. n.d. https://docs.espressif.com/projects/esp-idf/en/latest/esp32s3/hw-reference/esp32s3/user-guide-devkitm-1.html (Accessed November 7, 2023).

5 Adafruit Sensirion SHTC3 Temperature & Humidity Sensor STEMMA QT / Qwiic. Adafruit Industries, Unique & Fun DIY Electronics and Kits. n.d. https://www.adafruit.com/product/4636 (Accessed November 7, 2023).

6 Adafruit BMP390 Precision Barometric Pressure and Altimeter STEMMA QT / Qwiic. Adafruit Industries, Unique & Fun DIY Electronics and Kits. n.d. https://www.adafruit.com/product/4816 (Accessed November 7, 2023).

7 Adafruit VEML7700 Lux Sensor I2C Light Sensor STEMMA QT / Qwiic. Adafruit Industries, Unique & Fun DIY Electronics and Kits. n.d. https://www.adafruit.com/product/4162 (Accessed November 7, 2023).

8 Adafruit SGP30 Air Quality Sensor Breakout VOC and eCO2 STEMMA QT / Qwiic. Adafruit Industries, Unique & Fun DIY Electronics and Kits. n.d. https://www.adafruit.com/product/3709 (Accessed November 7, 2023).

9 Raspberry Pi Documentation - Raspberry Pi OS. n.d. https://www.raspberrypi.com/documentation/computers/os.html (Accessed November 7, 2023).

10 AWS IoT Core. n.d. https://aws.amazon.com/iot-core/ (Accessed November 7, 2023).

11 Python (programming language) – Wikipedia. n.d. https://en.wikipedia.org/wiki/Python_(programming_language) (Accessed November 7, 2023).

12 GitHub - aws/aws-iot-device-sdk-python-v2: Next Generation AWS IoT Client SDK for Python Using the AWS Common Runtime. n.d.

https://github.com/aws/aws-iot-device-sdk-python-v2#aws-iot-device-sdk-v2-for-python (Accessed November 7, 2023).

13 GitHub - blynkkk/lib-python: Blynk IoT library for Python and Micropython. n.d. https://github.com/blynkkk/lib-python#blynk-python-library (Accessed November 7, 2023).

14 What Is PlatformIO? — PlatformIO Latest Documentation. n.d. https://docs.platformio.org/en/latest/what-is-platformio.html (Accessed November 7, 2023).

15 GitHub - 256dpi/arduino-mqtt: MQTT library for Arduino. n.d. https://github.com/256dpi/arduino-mqtt (Accessed November 7, 2023).

16 GitHub - bblanchon/ArduinoJson: JSON Library for Arduino and Embedded C++. Simple and Efficient. n.d. https://github.com/bblanchon/ArduinoJson (Accessed November 7, 2023).

17 GitHub - adafruit/Adafruit_VEML7700: Arduino Library Driver for Adafruit VEML7700 Lux Sensor. n.d. https://github.com/adafruit/Adafruit_VEML7700 (Accessed November 7, 2023).

18 GitHub - adafruit/Adafruit-SGP30-PCB: PCB Files for The Adafruit SGP30 Air Quality Sensor. n.d. https://github.com/adafruit/Adafruit-SGP30-PCB (Accessed November 7, 2023).

19 GitHub - adafruit/Adafruit_BMP3XX. n.d. https://github.com/adafruit/Adafruit_BMP3XX (Accessed November 7, 2023).

20 GitHub - sparkfun/SparkFun_SHTC3_Arduino_Library: An Arduino Library to Take Humidity and Temperature Readings Using the SHTC3 Sensor from Sensirion. n.d. https://github.com/sparkfun/SparkFun_SHTC3_Arduino_Library (Accessed November 7, 2023).

Source Code

Source code for field devices

```
#include <Arduino.h>
#include "time.h"
#include "secrets.h"
#include <WiFiClientSecure.h>
#include <MQTTClient.h>
#include <ArduinoJson.h>
#include "WiFi.h"
#include "Adafruit_VEML7700.h"
#include <SPI.h>
#include <Adafruit_Sensor.h>
#include "Adafruit_BMP3XX.h"
#include <Wire.h>
#include "Adafruit_SGP30.h"
#include "SparkFun_SHTC3.h"
#include <Adafruit_BusIO_Register.h>

const unsigned long interval = 60000;  // Interval in milliseconds (1 minute)
unsigned long previousMillis = 0;


float sgpTVOC;
float sgpECO2;
float bmpAltitude;
float bmpPressure;
float luxData;
float airTemperature;
float airHumidity;
const char* ntpServer = "pool.ntp.org";
const long  gmtOffset_sec = 0;
const int   daylightOffset_sec = 10800;
void getLocalTime();
String formatTime(const struct tm* timeinfo);
const char* getMonthName(int month);
const char* getDayAbbreviation(int day);


#define WIRE_PORT   Wire    // This allows you to choose another Wire port (as
long as your board supports more than one)
SHTC3 mySHTC3;              // Declare an instance of the SHTC3 class

Adafruit_SGP30 sgp;

 uint32_t getAbsoluteHumidity(float temperature, float humidity) {
    const float absoluteHumidity = 216.7f * ((humidity / 100.0f) * 6.112f *
exp((17.62f * temperature) / (243.12f + temperature)) / (273.15f +
temperature));
    const uint32_t absoluteHumidityScaled = static_cast<uint32_t>(1000.0f *
absoluteHumidity);
    return absoluteHumidityScaled;
}

#define BMP_SCK 13
#define BMP_MISO 12
#define BMP_MOSI 11
#define BMP_CS 10

#define SEALEVELPRESSURE_HPA (1013.25)

Adafruit_BMP3XX bmp;

Adafruit_VEML7700 veml = Adafruit_VEML7700();
```

```
#define AWS_IOT_PUBLISH_TOPIC   "esp32-1/pub"
#define AWS_IOT_SUBSCRIBE_TOPIC "esp32-1/sub"

WiFiClientSecure wifi_client = WiFiClientSecure();
MQTTClient mqtt_client = MQTTClient(256);


void connectAWS()
{
  WiFi.mode(WIFI_STA);
  WiFi.begin(ssid, password);
  Serial.println("Connecting to WiFi service to send data to internet");
  int counter = 0;
  while (WiFi.status() != WL_CONNECTED){
    delay(1000);
    counter = counter + 1;
    if (counter >= 10){
        ESP.restart();
    }
  }
  Serial.println();
  Serial.println(WiFi.localIP());


  wifi_client.setCACert(AWS_CERT_CA);
  wifi_client.setCertificate(AWS_CERT_CRT);
  wifi_client.setPrivateKey(AWS_CERT_PRIVATE);

  mqtt_client.begin(AWS_IOT_ENDPOINT, 8883, wifi_client);


  Serial.print("Connecting to AWS cloud so data can be send");

  while (!mqtt_client.connect(THINGNAME)) {
    Serial.print(".");
    delay(100);
  }
  Serial.println();

  if(!mqtt_client.connected()){
    Serial.println("AWS IoT could not be connected, time might be out!");
    return;
  }

  mqtt_client.subscribe(AWS_IOT_SUBSCRIBE_TOPIC);

  Serial.println("AWS IoT Connected!");
}

void publishMessage()
{
  SHTC3_Status_TypeDef result = mySHTC3.update();

 if(mySHTC3.lastStatus == SHTC3_Status_Nominal)
  {
    Serial.print("Relative Humidity measured is: ");
    Serial.print(mySHTC3.toPercent
    airHumidity = mySHTC3.toPercent();
    Serial.print("% (checksum: ");
    if(mySHTC3.passRHcrc)
      Serial.print("checksum pass");
    }
    else
    {
      Serial.print("checksum fail");
    }
```

```
  Serial.print("), T = ");
  Serial.print(mySHTC3.toDegC());
  airTemperature = mySHTC3.toDegC();
  Serial.print(" deg C (checksum: ");
  if(mySHTC3.passTcrc)
  {
    Serial.print("pass");
  }
  else
  {
    Serial.print("fail");
  }
}
Serial.println("=======================================================");
Serial.println("=======================================================");
Serial.print("value of light intensity: ");
Serial.println(veml.readLux());
luxData = veml.readLux();
Serial.println("=======================================================");
Serial.println("=======================================================");

  if (! bmp.performReading()) {
  Serial.println("Failed to get data from bmp sensor :(");
  return;
}

Serial.print("value of pressure is:");
Serial.print(bmp.pressure / 100.0);
bmpPressure = bmp.pressure/100.0;
Serial.println(" hPa");

Serial.print("Approx. Altitude = ");
Serial.print(bmp.readAltitude(SEALEVELPRESSURE_HPA));
bmpAltitude = bmp.readAltitude(SEALEVELPRESSURE_HPA);
Serial.println(" meters.");

if (! sgp.IAQmeasure()) {
  Serial.println("Measurement failed");
  return;
}
Serial.print("TVOC ");
Serial.print(sgp.TVOC);
Serial.print(" parts per billion\t is:");
Serial.print("effective carbon dioxide is : ");
Serial.print(sgp.eCO2);
Serial.println(" parts per million.");
sgpTVOC = sgp.TVOC;
sgpECO2 = sgp.eCO2;
if (! sgp.IAQmeasureRaw()) {
  Serial.println("Raw Measurement failed");
  return;
}
Serial.print("hydrogen value is ");
Serial.print(sgp.rawH2);
Serial.print("Eth value is ");
Serial.print(sgp.rawEthanol);


  struct tm timeinfo;
   if(!getLocalTime(&timeinfo)){
  Serial.println("Failed to get timestamp from ntp server using internet");
  return;
}

StaticJsonDocument<200> doc;
doc["id"] = "ESP32-1";
doc["timestamp"] =  formatTime(&timeinfo);
```

```
  doc["airTemperature"] = airTemperature;
  doc["airHumidity"] = airHumidity;
  doc["luxData"] = luxData;
  doc["bmpPressure"] =  bmpPressure;
  doc["bmpAltitude"] =  bmpAltitude;
  doc["sgpTVOC"] = sgpTVOC;
  doc["sgpECO2"] = sgpECO2;
  char jsonBuffer[512];
  serializeJson(doc, jsonBuffer); // print to mqtt_client

  //Publish to the topic
  mqtt_client.publish(AWS_IOT_PUBLISH_TOPIC, jsonBuffer);
  Serial.println(" Data sent");
  delay(1000);
}

void incomingMessageHandler(String &topic, String &payload) {
  Serial.println("Message received!");
  Serial.println("Topic: " + topic);
  Serial.println("Payload: " + payload);
}

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

  while (!Serial)
    delay(10);     // will pause Zero, Leonardo, etc until serial console
opens

  connectAWS();
  configTime(gmtOffset_sec, daylightOffset_sec, ntpServer);
  getLocalTime();


  Serial.println("all hardware");

  if (!veml.begin()) {
    Serial.println("Sensor is not found, please check");
    while (1);
  }
  Serial.println("Sensor found");

  Serial.print(F("setting gain for sensor: "));
  switch (veml.getGain()) {
    case VEML7700_GAIN_1:
break;
    case VEML7700_GAIN_2:
break;
    case VEML7700_GAIN_1_4:
break;
    case VEML7700_GAIN_1_8:
break;
  }

  Serial.print(F("setting integration time for sensor: "));
  switch (veml.getIntegrationTime()) {
    case VEML7700_IT_25MS:
break;
    case VEML7700_IT_50MS:
break;
    case VEML7700_IT_100MS:
break;
    case VEML7700_IT_200MS:
break;
    case VEML7700_IT_400MS:
break;
    case VEML7700_IT_800MS:
```

```
break;
  }


  if (!bmp.begin_I2C())
    Serial.println("Could not find a BMP3 sensor, check if something is
wrong!");
    while (1);
  }

  bmp.setTemperatureOversampling(BMP3_OVERSAMPLING_8X);
  bmp.setPressureOversampling(BMP3_OVERSAMPLING_4X);
  bmp.setIIRFilterCoeff(BMP3_IIR_FILTER_COEFF_3);
  bmp.setOutputDataRate(BMP3_ODR_50_HZ);

    if (! sgp.begin()){
    Serial.println("Sensor seems to be not connected :(");
    while (1);
  }
  Serial.print("Found SGP30 serial #");
  Serial.print(sgp.serialnumber[0], HEX);
  Serial.print(sgp.serialnumber[1], HEX);
  Serial.println(sgp.serialnumber[2], HEX);

  WIRE_PORT.begin();

  Serial.print("Beginning sensor. Result = ");
  mySHTC3.begin(WIRE_PORT);
  Serial.println();

  if(mySHTC3.passIDcrc)
  {
    Serial.print("ID Passed Checksum. ");
    Serial.print("Device ID: 0b");
    Serial.print(mySHTC3.ID, BIN);
  }
  else
  {
    Serial.print("ID Checksum Failed. ");
  }
  Serial.println("\n");

  Serial.print("Choosing low-power measurements with T first: ");
  if(mySHTC3.setMode(SHTC3_CMD_CSE_TF_LPM) == SHTC3_Status_Nominal)
  {
    Serial.print("ok");
  }
  else
  {
    Serial.print(" not ok");
  }
  Serial.println("\n");

  Serial.print("Setting the sensor to stay awake at all times: ");
  if(mySHTC3.wake(true) == SHTC3_Status_Nominal)
  {
    Serial.print(" successful");
  }
  else
  {
    Serial.print(" failed");
  }
  Serial.println("\n\n");
}

void loop() {
 mqtt_client.loop();
```

```
 struct tm timeinfo;
  if (getLocalTime(&timeinfo)) {
    if (timeinfo.tm_sec == 0) {  // Check if the second value is 00
      // Code to send data goes here
      publishMessage();
    }
  }

}

void getLocalTime(){
  struct tm timeinfo;
  if(!getLocalTime(&timeinfo)){
    Serial.println("Failed to obtain time");
    return;
  }
  Serial.println(&timeinfo, "%A, %B %d %Y %H:%M:%S");
}



String formatTime(const struct tm* timeinfo) {
  char buffer[30];
  snprintf(buffer, sizeof(buffer), "%s %s %02d %02d:%02d:%02d %04d",
           getDayAbbreviation(timeinfo->tm_wday),
           getMonthName(timeinfo->tm_mon),
           timeinfo->tm_mday,
           timeinfo->tm_hour,
           timeinfo->tm_min,
           timeinfo->tm_sec,
           timeinfo->tm_year + 1900);
  return String(buffer);
}

// Function to get day abbreviation (e.g., Sun, Mon, Tue, ...)
const char* getDayAbbreviation(int day) {
  static const char* dayAbbreviations[] = {"Sun", "Mon", "Tue", "Wed", "Thu",
"Fri", "Sat"};
  return dayAbbreviations[day];
}

// Function to get month name (e.g., Jan, Feb, Mar, ...)
const char* getMonthName(int month) {
  static const char* monthNames[] = {"Jan", "Feb", "Mar", "Apr", "May", "Jun",
"Jul", "Aug", "Sep", "Oct", "Nov", "Dec"};
  return monthNames[month];
}
```

Source code for Master

```
from awscrt import io, mqtt, auth, http   ## sudo apt-get install cmake ##sudo
pip3 install awscrt
from awsiot import mqtt_connection_builder ## sudo pip3 install awsiotsdk
import json
import BlynkLib
from datetime import datetime, timedelta, date
import json

BLYNK_AUTH = 'DAq7N2_riSp7nZm8A562h3fjTvyRPc3m' #insert your Auth Token here
# base lib init
blynk = BlynkLib.Blynk(BLYNK_AUTH)

today = date.today()
timeToday = today.strftime("%d-%m-%Y")
filename = "/home/pi/Desktop/Data/"+timeToday+'.txt'


# Define ENDPOINT, CLIENT_ID, PATH_TO_CERTIFICATE, PATH_TO_PRIVATE_KEY,
PATH_TO_AMAZON_ROOT_CA_1, MESSAGE, TOPIC, and RANGE
ENDPOINT = "a1sai3943os71r-ats.iot.us-east-1.amazonaws.com"
CLIENT_ID = "Raspberry-Pi"
PATH_TO_CERTIFICATE = "/home/pi/blynk-library-python/deviceCertificate.crt"
PATH_TO_PRIVATE_KEY = "/home/pi/blynk-library-python/privateKey.key"
PATH_TO_AMAZON_ROOT_CA_1 = "/home/pi/blynk-library-python/amazonRootCA1.pem"

TOPIC = "esp32-1/pub"
TOPIC_2 = "esp32-2/pub"

received_data_esp32_1 = None
received_data_esp32_2 = None
timestamps_esp32_1 = None
timestamps_esp32_2 = None

event_loop_group = io.EventLoopGroup(1)
host_resolver = io.DefaultHostResolver(event_loop_group)
client_bootstrap = io.ClientBootstrap(event_loop_group, host_resolver)
mqtt_connection =
mqtt_connection_builder.mtls_from_path(endpoint=ENDPOINT,cert_filepath=PATH_TO
_CERTIFICATE,pri_key_filepath=PATH_TO_PRIVATE_KEY,client_bootstrap=client_boot
```

```
strap,ca_filepath=PATH_TO_AMAZON_ROOT_CA_1,client_id=CLIENT_ID,clean_session=F
alse,keep_alive_secs=6)


def process_and_calculate_average():
    global received_data_esp32_1, received_data_esp32_2, timestamps_esp32_1,
timestamps_esp32_2

    if (received_data_esp32_1 and received_data_esp32_2) or
(received_data_esp32_2 and received_data_esp32_1):
        time_difference = abs(timestamps_esp32_1 - timestamps_esp32_2)
        time_window = timedelta(seconds=15)
        time_window_max = timedelta(seconds=50)

        if time_difference <= time_window:
            # Calculate the average timestamp by taking the midpoint
            average_timestamp = timestamps_esp32_1 + (time_difference / 2)

            # Convert the average timestamp to a string in a desired format
            average_timestamp_str = average_timestamp.strftime("%Y-%m-%d
%H:%M:%S")

            # Calculate the average of all fields
            average_data = {
                "id": "avgData",
                "timestamp": average_timestamp_str,
                "airTemperature": (received_data_esp32_1["airTemperature"] +
received_data_esp32_2["airTemperature"]) / 2,
                "airHumidity": (received_data_esp32_1["airHumidity"] +
received_data_esp32_2["airHumidity"]) / 2,
                "luxData": (received_data_esp32_1["luxData"] +
received_data_esp32_2["luxData"]) / 2,
                "bmpPressure": (received_data_esp32_1["bmpPressure"] +
received_data_esp32_2["bmpPressure"]) / 2,
                "bmpAltitude": (received_data_esp32_1["bmpAltitude"] +
received_data_esp32_2["bmpAltitude"]) / 2,
                "sgpTVOC": (received_data_esp32_1["sgpTVOC"] +
received_data_esp32_2["sgpTVOC"]) / 2,
                "sgpECO2": (received_data_esp32_1["sgpECO2"] +
received_data_esp32_2["sgpECO2"]) / 2,
            }

            # Print the calculated average data
            print("Average Data:")
            print(average_data)
            json.dumps(average_data, indent=4)
            with open(filename, 'a+') as file:
                file.write(json.dumps(average_data))
                file.write("\n")
                file.flush()
                print('average data updated')
                print('=====================================')


            # Extract fields from average_data
            timestamp = average_data["timestamp"]
            air_temperature = average_data["airTemperature"]
            air_humidity = average_data["airHumidity"]
            lux_data = average_data["luxData"]
            bmp_pressure = average_data["bmpPressure"]
            bmp_altitude = average_data["bmpAltitude"]
            sgp_tvoc = average_data["sgpTVOC"]
            sgp_eco2 = average_data["sgpECO2"]

            # Send the average data to Blynk
            blynk.virtual_write(0, timestamp)
            blynk.virtual_write(1, air_temperature)
```

```
            blynk.virtual_write(2, air_humidity)
            blynk.virtual_write(3, lux_data)
            blynk.virtual_write(4, bmp_pressure)
            blynk.virtual_write(5, bmp_altitude)
            blynk.virtual_write(6, sgp_tvoc)
            blynk.virtual_write(7, sgp_eco2)
            # Reset received data and timestamps for the next round
            # received_data_esp32_1 = None
            # received_data_esp32_2 = None
            timestamps_esp32_1 = None
            timestamps_esp32_2 = None

        elif time_difference > time_window:
            print("Data received, but time window exceeded.")
            with open(filename, 'a+') as file:
                file.write("Data received, but time window maximum.")
                file.write("\n")
                file.flush()
                print('data updated')
                print('====================================')
                    # Reset received data and timestamps for the next round
            # received_data_esp32_1 = None
            # received_data_esp32_2 = None
            timestamps_esp32_1 = None
            timestamps_esp32_2 = None

        else:
            pass



def on_message_received(topic, payload, dup, qos, retain, **kwargs):
    global received_data_esp32_1, received_data_esp32_2, timestamps_esp32_1,
timestamps_esp32_2

    payloadToTxt = payload.decode()
    print(payloadToTxt)

    with open(filename, 'a+') as file:
        file.write(payloadToTxt)
        file.write("\n")
        file.flush()
        print('data updated')
        print('====================================')

    parsed_data = json.loads(payloadToTxt)
    if parsed_data["id"] == "ESP32-1":
        received_data_esp32_1 = parsed_data
        timestamps_esp32_1 = datetime.strptime(parsed_data["timestamp"], "%a
%b %d %H:%M:%S %Y")

    process_and_calculate_average()

def on_message_received_2(topic, payload, dup, qos, retain, **kwargs):
    global received_data_esp32_1, received_data_esp32_2, timestamps_esp32_1,
timestamps_esp32_2
    payloadToTxt = payload.decode()
    print(payloadToTxt)

    with open(filename, 'a+') as file:
        file.write(payloadToTxt)
        file.write("\n")
        file.flush()
        print('data updated')
        print('====================================')
```

```
    # Parse the JSON data
    parsed_data = json.loads(payloadToTxt)
    if parsed_data["id"] == "ESP32-2":
        received_data_esp32_2 = parsed_data
        timestamps_esp32_2 = datetime.strptime(parsed_data["timestamp"], "%a
%b %d %H:%M:%S %Y")

    process_and_calculate_average()

print("Connecting to {} with client ID '{}'...".format(
        ENDPOINT, CLIENT_ID))

if __name__ == '__main__':
    connect_future = mqtt_connection.connect()

    # Future.result() waits until a result is available
    connect_future.result()
    print("Connected!")

    # Subscribe
    print("Subscribing to topic '{}'...".format(TOPIC))
    subscribe_future, packet_id =
mqtt_connection.subscribe(topic=TOPIC,qos=mqtt.QoS.AT_LEAST_ONCE,callback=on_m
essage_received)

    subscribe_result = subscribe_future.result()
    print("Subscribed with code{} for
qos".format(str(subscribe_result['qos'])))

    # Subscribe to the second topic
    print("Subscribing to topic '{}'...".format(TOPIC_2))
    subscribe_future_2, packet_id_2 = mqtt_connection.subscribe(
        topic=TOPIC_2,
        qos=mqtt.QoS.AT_LEAST_ONCE,
        callback=on_message_received_2)

    subscribe_result_2 = subscribe_future_2.result()
    print("Subscribed to topic '{}' with {}".format(TOPIC_2,
str(subscribe_result_2['qos'])))


    while True:
        today = date.today()
        timeToday = today.strftime("%d-%m-%Y")
        filename = "/home/pi/Desktop/Data/"+timeToday+'.txt'
        blynk.run()
```