RTOS Based Implementation of Breath Data Collection Device with an Integrated 4G Cellular Data Logging Capability

Het Patel
November 30, 2023

MS Project
Intelligent Systems Lab
Department of Engineering Science
School of Science and Technology
Sonoma State University

_____
Dr. Sudhir Shrestha

_____
Dr. Mohamed Salem

_____
Dr. Nensung Wu

_____
Date

Table of Contents

List of Figures

CHAPTER 1: ABSTRACT

Purpose of the Project

The objective of this project is to develop a wireless automatic data logging device for breath based medical research. By utilizing an ESP32 microcontroller unit (MCU) with FreeRTOS operating system and a Notecarrier AF for cellular communication, the project aims to create a robust and reliable platform for remote breath data collection. The purpose is to provide an efficient solution that overcomes the limitations of previous iteration which utilized 3G protocol which has been phased out, having poor connectivity and is not usable anymore.

Results

The project successfully demonstrates the seamless transmission of breath-related data from the sensors to the cloud using 4G cellular communication. The system's multitasking capabilities, enabled by FreeRTOS, ensure efficient execution of tasks, allowing for real-time data collection. The implementation of the custom PCB significantly optimized the hardware architecture. It facilitated seamless integration of all on board peripherals, streamlining the system.

Conclusion

To conclude, the RTOS based breath data collection device development has taken thorough understanding and taken care of each minor component verification in firmware development. This project not only delved into the complexities of real-time task scheduling, sensor interfacing, and data acquisition but also necessitated multiple iterations of the PCB design to align with the project's objectives.

CHAPTER 2: INTRODUCTION

2.1 Motivation

The field of healthcare and medical technology has seen remarkable advancements in recent years, with an increasing emphasis on remote monitoring and data-driven insights. One critical aspect of this evolution is the acquisition of breath data for various diagnostic and analytical purposes. Traditional methods often rely on close-range communication protocols like WiFi or Bluetooth, limiting the range and reliability of data transmission. Additionally, the previous iteration was relying on the 3G data transmission cellular technology. In response to these limitations, the current project aims to develop a cutting-edge breath data monitoring system that leverages cellular communication for seamless data transfer over long distances.

The previous iteration of the system, developed by the ISL lab, was based on 3G technology. However, with the phase-out of 3G networks, the existing system became obsolete and inoperative. This necessitated the development of a new iteration utilizing 4G technology, ensuring continued operability and adaptability to evolving cellular networks. The challenge lies in seamlessly integrating the MCU with the Notecarrier-AF for reliable and efficient data transmission.

2.2 Project Overview

The aim of this project is to develop a 4G-based breath data monitoring system that leverages cellular communication for seamless and reliable data transmission over long distances. The ultimate goal is to develop the firmware using FreeRTOS and hardware in such a way that it will help the future iterations of the device when the other peripherals will be added to the device. This report will depict more information in the following chapters.

2.3 Block Diagram

The block diagram provides a comprehensive high-level overview of the entire system.

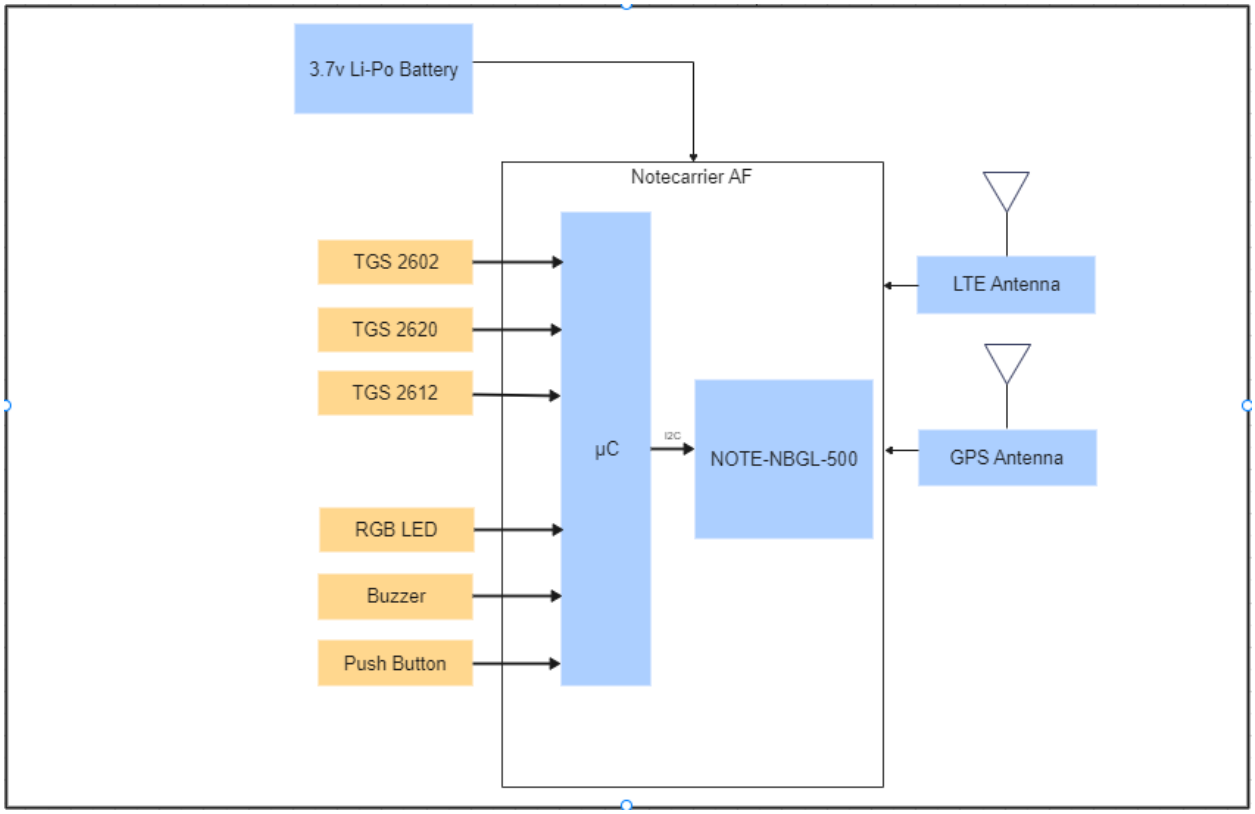

Fig 1. Simplified block diagram of the device

Please note that the mentioned block diagram is drawn based on the breadboard implementation. The following chapters will discuss how the device was customized while designing a PCB board.

CHAPTER 3: LITERATURE REVIEW

Various applications based on the Internet of Things (IoT) are employed for monitoring and controlling system parameters. Communication between modules and users is facilitated through wireless technologies like Wi-Fi, Bluetooth, RF, and ZigBee. While SCADA programs are developed for creating user interfaces, they are challenging to write and lack adaptability due to their cost and extensive libraries. Modules utilizing ZigBee, RF, and Bluetooth find wide applicability in user-friendly applications owing to their limited range, making them suitable for low data transfer rates. However, these communication techniques are constrained to simple applications due to their slow communication speeds, limited distances, and lower data security resulting from the absence of a full range of encryption techniques.

With the widespread availability of the internet, timer-controlled systems have seamlessly transitioned to remote-controlled systems. In these systems, obtaining information about both control and machine/device conditions via the internet is a crucial concern. To address this need, there have been initiatives to implement condition monitoring systems through the internet, leading to the development of internet-based remote control and monitoring practices.

3.1 Existing solutions:

Article: “Design and Implementation of ESP32-Based IoT Devices” [1]
The research paper article uses a Lilygo development board which consists of two major components; ESP32 as a SoC for all physical communication and SIM7000G for the cellular communication module. ESP32 is embedded on it that makes it easier for development and debugging during the process.

Limitations:

- Using an external SIM Card (has to be nano sim)
- Setup the SIM (Such as. APN details)
- Recurring charges (Approximately $10/month)

Article: “ESP32 Publish Data to Cloud without Wi-Fi (TTGO T-Call ESP32 SIM800L)”
The paper discusses how data transmission can occur when utilizing cellular connectivity. The on-board SIM800L - GSM/GPRS has been used for cellular communication. The External sim has been used which should have a data active data plan. Additionally, Need to know a few configuration details, APN and GPRS APN.

This can become a feasible solution when the project requirements are lower. When the cellular technology can be compromised and require everything on board, this could become the solution for the application.

[1]Design and Implementation of ESP32-Based IoT Devices
https://www.mdpi.com/1424-8220/23/15/6739
[2]ESP32 Publish Data to Cloud without Wi-Fi
https://randomnerdtutorials.com/esp32-sim800l-publish-data-to-cloud/

CHAPTER 4: HARDWARE DESIGN

4.1 List of Components

The project utilizes following parts:

The hardware design of this project builds upon the foundation laid by the previous iteration, a product of extensive research and development by the ISL team. This iteration incorporates key components that have proven to be highly effective in achieving the project's objectives. With the help of the previously involved team of ISL, the design was migrated from the previous iteration.

4.1.1 Notecarrier-AF

The central component in this design is the Blues Notecarrier-AF, a sophisticated module known for its reliable and robust 4G cellular communication capabilities. Its selection is a result of careful consideration, as it enables seamless data transmission over the cellular network.

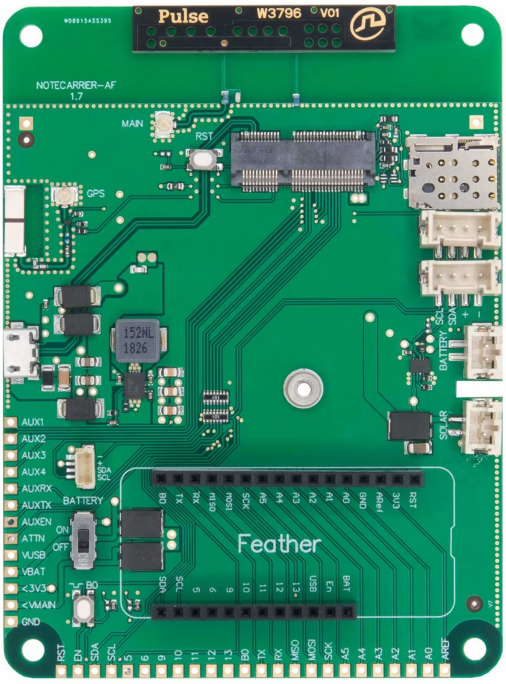

Fig 2. Notecarrier-AF 1.7

There have been several factors which are associated while choosing this module over other similar modules. These offers below functionalities:

- Support 4G connectivity
- Simple to write firmware
- Easy to call open source libraries
- Support range of MCUs, specifically ESP32 based
- One time fixed cost for 10 years
- Lower voltage requirement
- Compatible with 3.3v or 5v equipment

4.1.2 Notecard

The Notecard is a pivotal standalone component in this project, serving as the linchpin for cellular communication and data transmission. A SoM peripheral designed to bridge the gap between prototype and production for the Notecard. The notecarrier AF supports range of notecards and each differs from their size and the functionality it provides:

- NOTE-NBGL-500
- NOTE-WBNA-500
- NOTE-NBNA-500
- NOTE-WBEX-500
- NOTE-WIFI
- NOTE-ESP
- NOTE-LORA

However, this project utilizes Notecard version- NOTE-WBNA-500 where GL stands for Global and NB stands for Narrow Band. The Data network used by Notecard is LTE-M / NB-IoT / GPRS. The reason for choosing this module over the rest of the variants is because this version satisfies the need of sending data to the cloud.

The Notecard is designed to be socketed directly onto the circuit board using an edge connector socket which enables the ability to handle data transmission over cellular networks with remarkable efficiency.

The Notecard is connected to the Notecarrier using the Universal Asynchronous Receiver-Transmitter (UART) interface. The UART protocol allows for serial communication between the Notecard and the microcontroller (in this case, the Adafruit ESP32 Huzzah). This two-wire interface, Tx for transmission and Rx for receiver, enables the exchange of data, allowing the microcontroller to send commands and receive responses from the Notecard.
Notecard Data Transmission Architecture

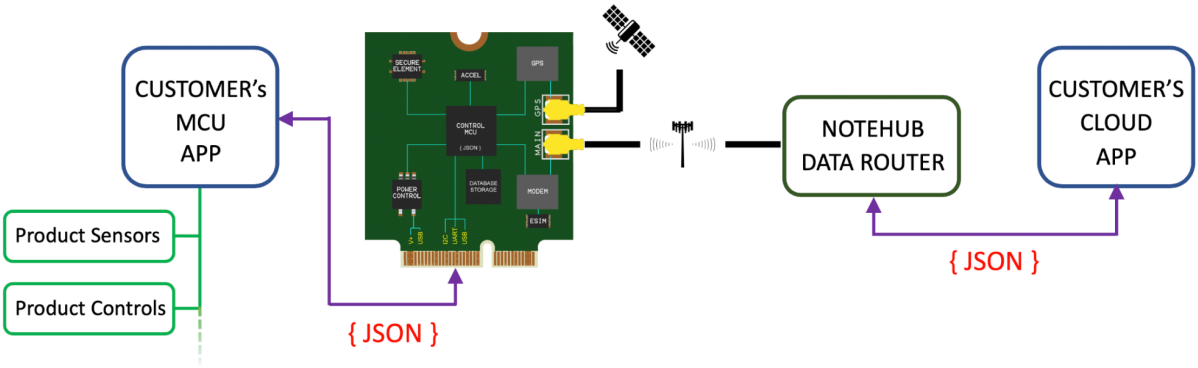

Fig 3. Flow of data transmission

The above diagram depicts the end to end transmission of the data from user to the cloud side. Note that the data transmission has been based on the JSON format.
Advantages using a JSON:

- **Lightweight**: it mainly contains the data that needs to be transferred, rather than a lot of markup used to define the structure of the data.
- **Concise and Compact:** Compared to other languages like XML, JSON makes it faster to parse and generate. This reduces the size and bandwidth of data transfers, as well as improving the performance and efficiency of data processing.
- **Readability**: It is easier to read and write than XML, which improves code readability and maintainability.

Below is an example of JSON data where note.add. This request adds a single Note with an arbitrary body or payload to a Notefile, which is an outbound queue of data to synchronize to Notehub.. They mainly work on key value pairs format utilized by JSON.

```
> {"req":"note.add","body":{"temp":35.5,"humid":56.23}}
{"total":1}
```

Fig 4. Example of JSON data

4.1.3 Microcontroller:

A low cost, low-cost embeddable ESP32 based microcontroller compatibility with the Notecard and NotecarrierAF influenced the selection for this project. This compatibility includes dedicated libraries, making it straightforward to set up and use the MCU. Additionally, the ability to place MCU on the socket pin headers was matching with the MUC pins.

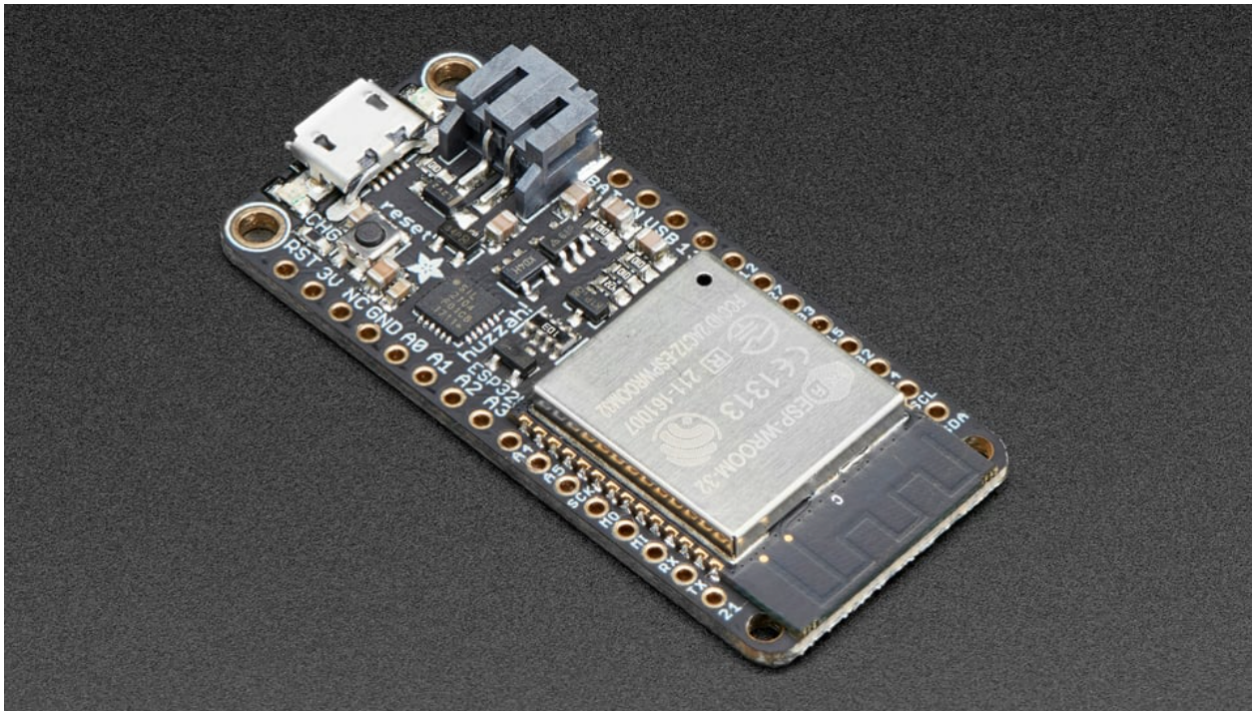

Fig 5. Adafruit ESP32 Hazzah

The MCU is divided into a total of two components, a dual-core ESP32 chip and the rest of the components surrounding the chip. Those components contain, ADC and DAC converter, MicroUSB port, LiPo battery connector, a reset button, etc.

4.1.4 LiPo Battery:

Both Notecarreir and MCU are designed to work with voltages ranging from 3.3V to 5V, making the Li-Po battery an ideal power source for our system The Li-Po battery supplies a voltage of 3.7V, which falls within the operational range of both the Notecarrier and the MCU. The battery is pre-attached with a 2-pin JST-PH connector.

This battery has a capacity of 1200mAh. There is more detailed information on current consumption in chapter x.

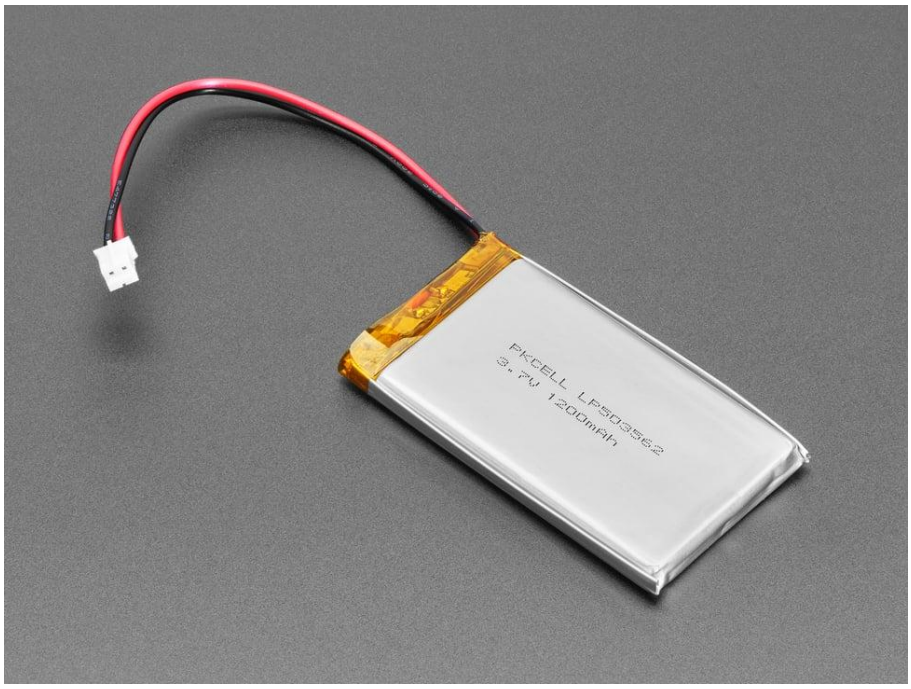

Fig 6. External 3.7v LiPo battery

Chapter 5: Firmware Development

The firmware for this project was developed using the Arduino Integrated Development Environment (IDE). The Arduino IDE is a cross-platform application. It is used to write and upload programs to Arduino compatible boards, but also, with the help of third-party cores, other vendor development boards.

The choice of the ESP32 microcontroller was due to the ability to adapt to Notecarrier, its powerful features such as multiple GPIOs (General Purpose Input Output), and compatibility with the Arduino IDE. The ESP32 is a series of low-cost, low-power systems on a chip microcontroller. The ESP32 series employs a Tensilica Xtensa LX6 microprocessor in both dual-core and single-core variations and includes built-in antenna switches, RF balun, power amplifier, low-noise receive amplifier, filters, and power-management modules.

The firmware leverages the FreeRTOS operating system. FreeRTOS is a real-time operating system kernel for embedded devices that has been ported to multiple microcontroller platforms. It is distributed under the MIT License. The ‘RTOS’ part of the name is an abbreviation for ‘Real-Time Operating System’. This is a type of program that is designed to manage the resources of a computer or processor in real-time. Typically, this involves implementing features such as multi-tasking, process or thread scheduling, and memory management.

The firmware architecture of this project is designed around the FreeRTOS real-time operating system, which allows for efficient multitasking on the ESP32 microcontroller. The firmware is divided into three main tasks, each responsible for a specific part of the program. Here’s a detailed breakdown:

5.1 Firmware Tasks

5.1.1 Sensor Reading Task

This task is responsible for reading data from the TGS2602, TGS2612, and TGS2620 sensors. These sensors are gas sensors that are sensitive to different types of gases. The TGS2602 sensor is sensitive to air contaminants such as hydrogen and volatile organic compounds (VOCs). These sensors are warmed up for 4 minutes at startup, after which they are ready to take breath input and generate digital readings. The readings are taken three times and stored in an array.

This task starts by turning on the power to the sensors and allowing them to warm up. During this warm-up period, an LED is flashed to indicate that the sensors are getting ready. Once the sensors are ready, the task enters a loop where it continuously checks for a button press. If the button is pressed, the task reads the sensor values and stores them in an array.

Below is the detailed explanation of the Sensor reading task implementation in the firmware.

```
void vTask1(void *pvParameters) {
    unsigned long lastMillis = 0;
    unsigned long task1StartTime = 0;
    unsigned long task1EndTime = 0;
    ArrayData arrayData;

    ulTaskNotifyTake(pdTRUE, portMAX_DELAY);

    digitalWrite(SENS_POWER, HIGH);
    lastMillis = millis()/1000;

    while ((millis()/1000-lastMillis) < (SENS_WARMING_TIME)) {
      digitalWrite(R_LED, HIGH);
      delay(250);
      digitalWrite(R_LED, LOW);
      delay(600);
      checkLongPress();
      delay(10);
    }

    while(1)
    {
      for (int i = 0; i < 3; i++) {
        sensor1_2612A[i] = analogRead(SENSOR1_2612);
        delay(10);
        sensor2_2602A[i] = analogRead(SENSOR2_2602);
        delay(10);
        sensor3_2620A[i] = analogRead(SENSOR3_2620);
        delay(500);
      }

      lastMillis = millis();
      int i = 0, j = 0;

      while (digitalRead(BUTTON_PIN)) {
        digitalWrite(R_LED, HIGH);
        delay(100);
        digitalWrite(R_LED, LOW);
        delay(100);
        tone(BUZZ_PIN,457,250);//Sound the buzzer
        delay(100);

      if ((millis()-lastMillis)>1000){
        if (i > 2) {
          i = 0;
```

```
        }
        sensor1_2612A[i] = analogRead(SENSOR1_2612); //reads sensor1 input for
air
        delay(10);
        sensor2_2602A[i] = analogRead(SENSOR2_2602); //reads sensor2 input for
air
        delay(10);
        sensor3_2620A[i] = analogRead(SENSOR3_2620); //reads sensor3 input for
air
        delay(10);
        i++;
        lastMillis = millis();
        j++;
      }
       if (j > BEEP_TIME){
         Low_Power_Sleep();
      }
    }
    digitalWrite(R_LED, LOW); //turn LED off
    delay(10);

    for (int i = 0; i < 20; i++) {
      digitalWrite(G_LED, HIGH);
      delay(100);
      digitalWrite(G_LED, LOW);
      delay(100);
   }

    for (int i = 0; i < 3; i++) {
      sensor1_2612B[i] = analogRead(SENSOR1_2612);
      delay(10);
      sensor2_2602B[i] = analogRead(SENSOR2_2602);
      delay(10);
      sensor3_2620B[i] = analogRead(SENSOR3_2620);
      delay(10);

    for (int i = 0; i < 5; i++) {
      digitalWrite(G_LED, HIGH);
      delay(100);
      digitalWrite(G_LED, LOW);
      delay(100);
    }
  }

  for (int i = 0; i < 3; i++) {
```

```
  digitalWrite(R_LED, HIGH);
  delay(100);
  digitalWrite(R_LED, LOW);
  delay(100);
  tone(BUZZ_PIN,457,100);//Sound the buzzer
  delay(50);
 }

 lastMillis = millis();
 int second_read = 1;
 int skip_second_read = 0;

while (digitalRead(BUTTON_PIN) && second_read) {
  digitalWrite(R_LED, HIGH);
  delay(100);
  digitalWrite(R_LED, LOW);
  delay(300);

  if (second_read && (((millis()-lastMillis)/1000)>60)){
    second_read = 0;
    skip_second_read = 1;

   for (int i = 3; i < 6; i++) {
    sensor1_2612B[i] = 0;
    delay(10);
    sensor2_2602B[i] = 0;
    delay(10);
    sensor3_2620B[i] = 0;
    delay(10);
    }
  }
}

if(!skip_second_read){
   for (int i = 0; i < 20; i++) {
    digitalWrite(G_LED, HIGH);
    delay(100);
    digitalWrite(G_LED, LOW);
    delay(100);
   }

 for (int i = 3; i < 6; i++) {
  sensor1_2612B[i] = analogRead(SENSOR1_2612); //reads sensor1 input for air
  delay(10);
  sensor2_2602B[i] = analogRead(SENSOR2_2602); //reads sensor2 input for air
```

```
  delay(10);
  sensor3_2620B[i] = analogRead(SENSOR3_2620); //reads sensor3 input for air
  delay(10);

 for (int i = 0; i < 5; i++) {
  digitalWrite(G_LED, HIGH);
  delay(100);
  digitalWrite(G_LED, LOW);
  delay(100);
  }
 }
}

if(!skip_second_read){
   for (int i = 0; i < 3; i++) {
    digitalWrite(R_LED, HIGH);
    delay(100);
    digitalWrite(R_LED, LOW);
    delay(100);
    tone(BUZZ_PIN,457,100);//Sound the buzzer
    delay(50);
   }
}

digitalWrite(SENS_POWER, LOW);
delay(10);

digitalWrite(G_LED, LOW);
delay(10);
digitalWrite(R_LED, LOW);
delay(10);

arrayData.VOL_ADC_VALUE[0] = sensor1_2612B[0];
arrayData.VOL_ADC_VALUE[1] = sensor1_2612B[1];
arrayData.VOL_ADC_VALUE[2] = sensor1_2612B[2];
arrayData.VOL_ADC_VALUE[3] = sensor1_2612B[3];
arrayData.VOL_ADC_VALUE[4] = sensor1_2612B[4];
arrayData.VOL_ADC_VALUE[5] = sensor1_2612B[5];

arrayData.VOL_ADC_VALUE[6] = sensor2_2602B[0];
arrayData.VOL_ADC_VALUE[7] = sensor2_2602B[1];
arrayData.VOL_ADC_VALUE[8] = sensor2_2602B[2];
arrayData.VOL_ADC_VALUE[9] = sensor2_2602B[3];
arrayData.VOL_ADC_VALUE[10] = sensor2_2602B[4];
arrayData.VOL_ADC_VALUE[11] = sensor2_2602B[5];
```

```
  arrayData.VOL_ADC_VALUE[12] = sensor3_2620B[0];
  arrayData.VOL_ADC_VALUE[13] = sensor3_2620B[1];
  arrayData.VOL_ADC_VALUE[14] = sensor3_2620B[2];
  arrayData.VOL_ADC_VALUE[15] = sensor3_2620B[3];
  arrayData.VOL_ADC_VALUE[16] = sensor3_2620B[4];
  arrayData.VOL_ADC_VALUE[17] = sensor3_2620B[5];

  xQueueSend(xQueue, &arrayData, portMAX_DELAY);
  task1EndTime = millis();

  Serial.println("Task 1 Execution Time: ");
  Serial.println(task1EndTime - task1StartTime);

  ulTaskNotifyTake(pdTRUE, portMAX_DELAY);
  }
}}
```

The above vTask1 function is a task in the FreeRTOS system that handles the sensor reading and data preparation for transmission. It's divided into several sections:

1. *Initialization:* The function starts by initializing several variables used to track time and store sensor readings. It then waits for a notification to start the task.

2. *Sensor Warmup:* The function turns on the power to the sensors and enters a loop where it flashes an LED to indicate that the sensors are warming up. This loop continues until the sensors have warmed up for a specified amount of time.

3. *First Sensor Reading*: After the sensors have warmed up, the function enters an infinite loop where it continuously reads sensor values and checks for a button press. If the button is pressed, it reads the sensor values three times and stores them in an array. Note that the utilized ADC resolution is 12 bit which returns a value from 0 to 4095 and stored in the array.

4. *Second Sensor Reading:* If the button is pressed for a long time, the function enters a loop where it reads the sensor values again and stores them in a different array. This loop continues until a certain condition is met.

5. *Data Preparation:* After the sensor readings have been taken, the function stores the readings in a structure and sends the structure to a queue for transmission. It then calculates the execution time of the task and waits for another notification to start the next cycle.

5.1.2 JSON Packet Creation Task

Once the sensor readings have been captured, this task converts the readings into a JSON format. JSON (JavaScript Object Notation) is a popular data format with diverse uses in data interchange, including that of web applications with servers. JSON is a language-independent data format. It was derived from JavaScript, but many modern programming languages include code to generate and parse JSON-format data. JSON filenames use the extension. json.

Below is the detailed explanation of the JSON Packet Creation task implementation in the firmware.

```
J *req = NULL;
The J *req = NULL; line declares a pointer to a J object, which is used to
create JSON requests.
void vTask2(void *pvParameters) {
    unsigned long task2StartTime = 0;
    unsigned long task2EndTime = 0;
    ArrayData arrayData;
    while (1) {
        task2StartTime = millis();
        float voltage;
        if (xQueueReceive(xQueue, &arrayData, portMAX_DELAY) == pdTRUE) {
            float temperature = sensor.temp();
            float humidity = sensor.humidity();

            req = notecard.newRequest("note.add");
            if (req != NULL) {
                JAddStringToObject(req, "file", "sensors.qo");
                JAddBoolToObject(req, "sync", true);
                J *body = JAddObjectToObject(req, "body");
                if (body) {
                    JAddNumberToObject(body, "temp", temperature);
                    JAddNumberToObject(body, "humidity", humidity);
                    JAddNumberToObject(body, "Sensor 1_1",
arrayData.VOL_ADC_VALUE[0]);
                    JAddNumberToObject(body, "Sensor 1_2",
arrayData.VOL_ADC_VALUE[1]);
                    JAddNumberToObject(body, "Sensor 1_3",
arrayData.VOL_ADC_VALUE[2]);
                    JAddNumberToObject(body, "Sensor 1_4",
arrayData.VOL_ADC_VALUE[3]);
                    JAddNumberToObject(body, "Sensor 1_5",
arrayData.VOL_ADC_VALUE[4]);
```

```
                    JAddNumberToObject(body, "Sensor 1_6",
arrayData.VOL_ADC_VALUE[5]);
                    JAddNumberToObject(body, "Sensor 2_1",
arrayData.VOL_ADC_VALUE[6]);
                    JAddNumberToObject(body, "Sensor 2_2",
arrayData.VOL_ADC_VALUE[7]);
                    JAddNumberToObject(body, "Sensor 2_3",
arrayData.VOL_ADC_VALUE[8]);
                    JAddNumberToObject(body, "Sensor 2_4",
arrayData.VOL_ADC_VALUE[9]);
                    JAddNumberToObject(body, "Sensor 2_5",
arrayData.VOL_ADC_VALUE[10]);
                    JAddNumberToObject(body, "Sensor 2_6",
arrayData.VOL_ADC_VALUE[11]);
                    JAddNumberToObject(body, "Sensor 3_1",
arrayData.VOL_ADC_VALUE[12]);
                    JAddNumberToObject(body, "Sensor 3_2",
arrayData.VOL_ADC_VALUE[13]);
                    JAddNumberToObject(body, "Sensor 3_3",
arrayData.VOL_ADC_VALUE[14]);
                    JAddNumberToObject(body, "Sensor 3_4",
arrayData.VOL_ADC_VALUE[15]);
                    JAddNumberToObject(body, "Sensor 3_5",
arrayData.VOL_ADC_VALUE[16]);
                    JAddNumberToObject(body, "Sensor 3_6",
arrayData.VOL_ADC_VALUE[17]);
                }
                 Serial.println("Task 2 Executing");
                 task2EndTime = millis();
                 Serial.println("Task 2 Execution Time: ");
                 Serial.println(task2EndTime - task2StartTime);
                 Serial.println("ms");
                 xTaskNotify(xTask3Handle, 0x01, eSetBits); // Send a
notification to task 3 to start execution
            }
        }
    }
}
```

The above vTask2 function is a task in the FreeRTOS system that handles the creation of a JSON packet from the sensor readings. It's divided into several sections:

1. *Initialization:* The function starts by initializing several variables used to track time and store sensor readings. It then enters an infinite loop where it continuously checks for data in the queue.

2. *Data Retrieval:* If data is available in the queue, the function retrieves the data and calculates the temperature and humidity.

3. *JSON Packet Creation:* The function then creates a new JSON request using the notecard.newRequest function. If the request is successfully created, it adds several key-value pairs to the request, including the file name, sync status, and sensor readings. These key-value pairs are added using the JAddStringToObject, JAddBoolToObject, and JAddNumberToObject functions.

4. *Execution Time Calculation and Task Notification:* After the JSON packet has been created, the function calculates the execution time of the task and sends a notification to the Data Sending Over Cloud Task (vTask3) to indicate that the JSON packet is ready to be sent.

5.1.3 Data Sending Over Cloud Task

This task is responsible for sending the JSON packet over the cloud. The packet is sent to a Notecard device, which then transmits the data to the cloud. The Notecard is a small, low-cost IoT device for sending data to the cloud. It includes built-in cellular service for global connectivity right out of the box. It's perfect for a wide range of IoT applications, including asset tracking, remote monitoring, and smart city solutions.

```
void vTask3(void *pvParameters) {
    unsigned long task3StartTime = 0; // Declare a variable to hold the start
time of task 3
    unsigned long task3EndTime = 0; // Declare a variable to hold the end time
of task 3
    while (1) { // Loop indefinitely
        task3StartTime = millis(); // Get the current time in milliseconds
        ulTaskNotifyTake(pdTRUE, portMAX_DELAY); // Wait for a notification to
start execution

        notecard.sendRequest(req); // Send the Notecard request created in task
2

        digitalWrite(R_LED, HIGH); // Turn on the red LED
        digitalWrite(G_LED, HIGH); // Turn on the green LED
        delay(2000); // Wait for 2 seconds
        digitalWrite(R_LED, LOW); // Turn off the red LED
        digitalWrite(G_LED, LOW); // Turn off the green LED

        Serial.println("--> Data Logging to SD Card");
```

```
        //Log data to the SD Card - Added by Natalie and Jessie, Aug 2, 2022
        if(SD.begin(chipSelect)){
          // open the file. note that only one file can be open at a time,
          // so you have to close this one before opening another.
          DataFile = SD.open("data.txt", FILE_WRITE);
          if (DataFile) {
            DataFile.print("\n");
            DataFile.print("PData");//indicates patient data -
            DataFile.print(", ");
            DataFile.print(", ");
            DataFile.print(arrayData.VOL_ADC_VALUE[0]); //air(blank) read data
            DataFile.print(", ");

            for(int i=0; i<27; i++){
              DataFile.print(arrayData.VOL_ADC_VALUE[i]); //air(blank) read
data

              DataFile.print(", ");
            }
            // close the file:
            DataFile.close();
            }
            else {
              Serial.println("Error opening data.txt");
            }
        }
        else {
          Serial.println("Error initializing SD card");
        }

        //Code for checking SD card write
        // re-open the file for reading:
        DataFile = SD.open("data.txt");
        if (DataFile) {
          Serial.println("data.txt:");

          // read from the file until there's nothing else in it:
          while (DataFile.available()) {
            Serial.write(DataFile.read());
          }
          // close the file:
          DataFile.close();
        } else {
          // if the file didn't open, print an error:
          Serial.println("error opening data.txt");
        }
```

```
        task3EndTime = millis(); // Get the current time in milliseconds

        Serial.println("Task 3 Execution Time: ");
        Serial.println(task3EndTime - task3StartTime);
        Serial.println("ms");

        xTaskNotify(xTask1Handle, 0x01, eSetBits); // Send a notification to
task 1 to start execution
    }
}
```

The vTask3 function is a task in the FreeRTOS system that handles the sending of the JSON packet over the cloud. It's divided into several sections:

1. *Initialization:* The function starts by initializing several variables used to track time. It then enters an infinite loop where it continuously waits for a notification to start the task.

2. *Data Transmission:* Once it receives a notification, it sends the JSON packet to the Notecard device using the notecard.sendRequest(req) function.

3. *LED Indication:* After sending the request, it turns on both the red and green LEDs for 2 seconds to indicate successful transmission, and then turns them off.

4. *Execution Time Calculation and Task Notification:* The function then calculates the execution time of the task and sends a notification back to the Sensor Reading Task (vTask1) to indicate that it's ready for the next set of sensor readings.

5.1.4 Setup Function

The setup function initializes the system:

```
void setup() {
  // put your setup code here, to run once:
    delay(2500); // Wait for 2.5 seconds
    usbSerial.begin(115200); // Initialize the USB serial port with a baud rate
of 115200

    pinMode(SENS_POWER, OUTPUT); // Set the sensor power pin as an output
    pinMode(R_LED, OUTPUT); // Set the red LED pin as an output
    pinMode(G_LED, OUTPUT); // Set the green LED pin as an output

    notecard.begin(); // Initialize the Notecard
    notecard.setDebugOutputStream(usbSerial); // Set DebugOutputStream method,
```

```
which will output raw JSON requests and responses.

    xQueue = xQueueCreate(30, sizeof(ArrayData)); // Create a queue to hold 10
elements of type ArrayData

    xTaskCreate(vTask1, "Task 1", 2000, NULL, 1, &xTask1Handle); // Create task
1 with a stack size of 2000 bytes and priority 1
    xTaskCreate(vTask2, "Task 2", 2000, NULL, 1, &xTask2Handle); // Create task
2 with a stack size of 2000 bytes and priority 1
    xTaskCreate(vTask3, "Task 3", 2000, NULL, 1, &xTask3Handle); // Create task
3 with a stack size of 2000 bytes and priority 1

    xTaskNotify(xTask1Handle, 0x01, eSetBits); // Send a notification to task 1
to start execution
}
```

This function is called once when the program starts. It initializes the serial communication, sets the pin modes, starts the Notecard device, creates a queue for data transfer, creates three tasks (vTask1, vTask2, and vTask3), and sends a notification to vTask1.

5.1.5 Loop Function

The loop function is empty because all the tasks are handled by the FreeRTOS system and taken care of in the setup task.

```
void loop() {
}
```

Chapter 6: Application Flow

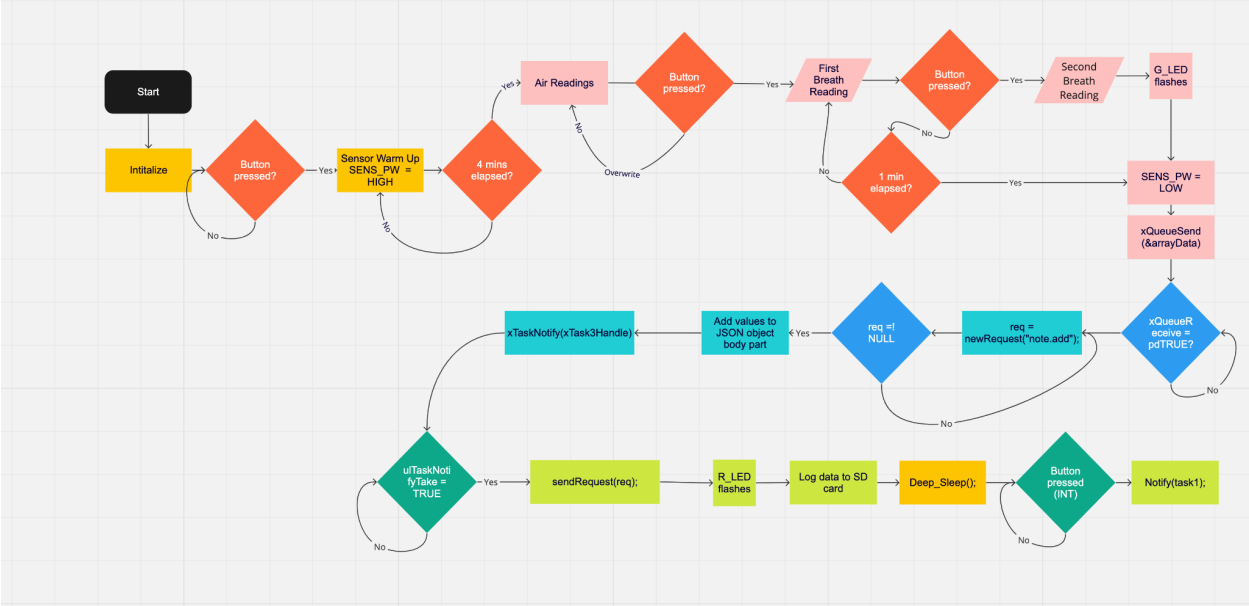

Fig.8 Detailed view of the firmware

Startup and Sensor Warmup

At startup, the transistors controlling the VCC of the sensors are turned on, and the sensors enter a warmup state. During this time, the sensors receive continuous current for 4 minutes to heat up. This is necessary because the sensors need to reach a certain temperature to operate correctly. The warmup period ensures that the sensors are at the right temperature before they start taking readings.

LED Indication for Sensor Readiness

After the sensors have warmed up, an LED controlled by the ESP32 flashes three times. This indicates that the sensors are ready to take breath input. LEDs (Light Emitting Diodes) are used in this project as indicators due to their low power consumption and long life.

Breath Input Acquisition and Reading Process

During the input acquisition period, the buzzer is turned on, and the reading acquisition process is completed. This process is repeated three times, and the readings are printed on the serial monitor. The serial monitor is a feature of the Arduino IDE that allows text data to be sent and received by an Arduino and a computer.

Conversion of Readings to JSON Format and Cloud Transmission

The readings are then converted into a JSON format and sent to the cloud. The LED is flashed, and the buzzer is turned on for 2 seconds to indicate successful transmission.

Chapter 7: PCB development

The PCB development for the Real-time Air Quality Monitoring System played a pivotal role in achieving the project's objectives. The PCB served as a crucial component, integrating various hardware elements and providing the foundation for the device's functionality.

The PCB design passes through three significant iterations, driven by the necessity to optimize the device's size for enhanced user-friendliness. The revisions were made few amount of time to match the design requirements.

Altium Designer software has been used through the design process. The software is made industrial standard and provides a significant amount of functionalities to customize the design board. Altium Designer stands out as a comprehensive and powerful software tool extensively used in the field of PCB (Printed Circuit Board) development. Overall, Altium Designer's robust set of features and user-friendly interface make it a popular choice among PCB designers.

The design development process goes through several stages which are listed below:

- Schematic symbol design
- Parts footprint creation
- Board layout
- Routing

Initial design files were imported from Blue's Github Repository which includes the Notecarrier and Notecard design. This has helped to make the design process straightforward and gave the initial boost.

7.1 Schematic symbol design

The initial process started from finding the symbol for most of the peripherals components and designing custom for the same. The example mentioned below. The full schematic design has been added in the appendix.

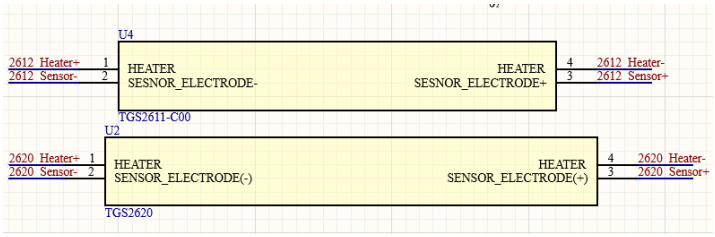

Fig 9. Symbol design for schematic presentation

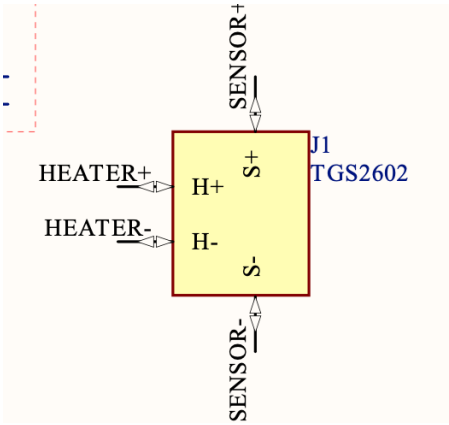

Fig 10. Symbol design for schematic presentation

7.2 Footprint selection

Appropriate footprint selection is a crucial aspect of the PCB design process, involving the careful choice of physical layouts for each component on the board. The footprint, essentially a pattern of solder pads and connecting traces, must align with the specific dimensions and specifications of the component it represents.

If a wrong footprint is used for a component on a printed circuit board (PCB), it can lead to various issues, ranging from manufacturing and assembly problems to operational failures and can even lead to failure in the prototype functioning.

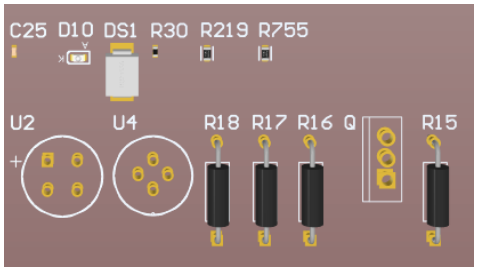

Fig 11. Footprint representation

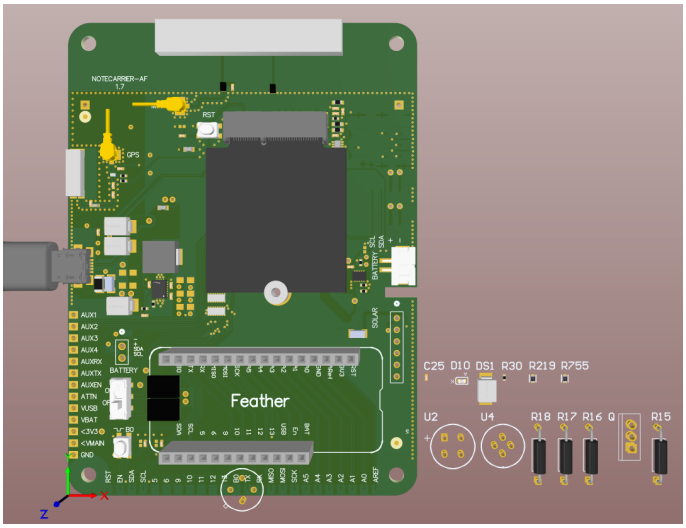

Fig 12. The initial component placement

As mentioned above, the left board has been imported from the Blue's repository design. The components footprint was imported from renowned sources like SnapEDA and manufacturer websites.

The following snapshot shows the redesigned version while considering the design requirements.

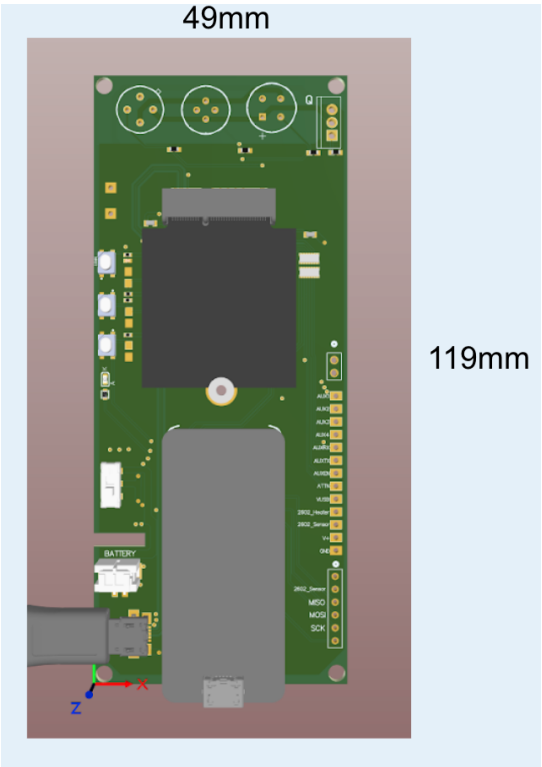

Fig 13. Redesigned version 1

Note that the antennas are removed from the design. After considering a few suggestions, I decided to remove the antennas on board and can be added externally connecting directly from the Notecard. This will provide extra space to append sensors on top.

Due to the various on-board antennas which are designed in small space resulting in the performance degradation of the overall circuit.

Subsequently, the initial design iteration proved to be imperfect due to concerns regarding the device's height. Eventually, a refinement was undertaken, resulting in a reduction of the height to 10cm and width to 5cm while the on-board components were strategically repositioned to enhance the overall design. Following the snapshot that proves the changes:

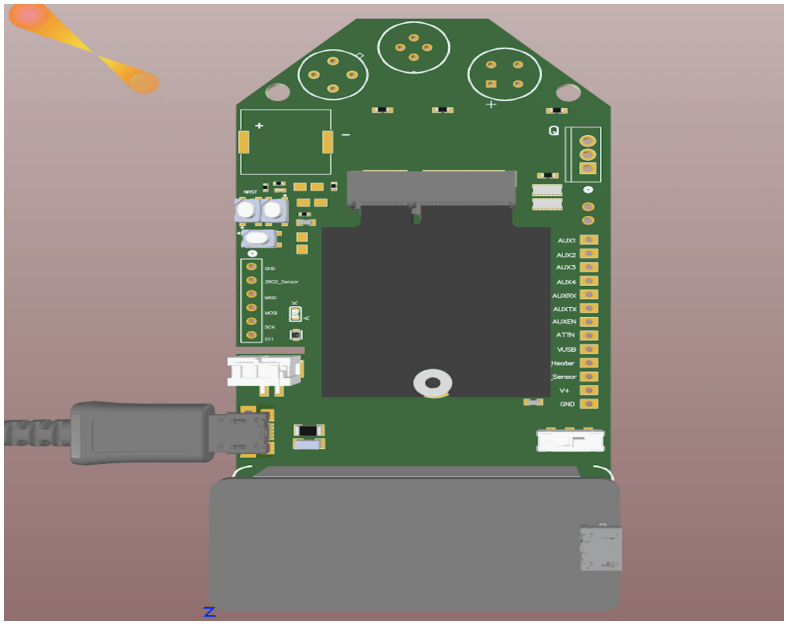

Fig 14. The final PCB design

Chapter 9: Results and Discussion

The following images depicts the overall results and output of this project after performing the test. Fig. x shows the serial output when the program starts executing.

```
------------Welcome to ISL Research Study Prototype------------
-------------------------Program Started-------------------------
--------------------Sensor Power has been Fed-------------------
Button is pressed inside checkLongPress function
Elapsed time: 2 seconds
Elapsed time: 3 seconds
Elapsed time: 4 seconds
Elapsed time: 5 seconds
Elapsed time: 5 seconds
Elapsed time: 6 seconds
Elapsed time: 7 seconds
Elapsed time: 8 seconds
Elapsed time: 9 seconds
Elapsed time: 10 seconds
Elapsed time: 11 seconds
Elapsed time: 12 seconds
Elapsed time: 12 seconds
Elapsed time: 13 seconds
```

Fig 15. Notehub data representation

This is the phase when the initial part of task 1 is completed (sensor warm up period) and now about to take the air readings and follow up breath readings.

```
--> Sensor warming completed
--> First AIR reading
E (46254) ledc: ledc_get_duty(739): LEDC is not initialized
--> First BREATH Reading
--> Taken
--> Waiting for the button to press, Second Breath
--> Reading Sensors NOW last time, Second Breath Reading
Test Completed Succefully
```

Fig 16. Notehub data representation

The fig x. shows the final output, which is called as a note, that has been sent out to the notehub cloud platform and further will be routed to the Azure cloud data central. It is clear that the program was successfully able to perform each of the mentioned operations and able to send the requested data to the cloud.

{"req":"note.add","file":"sensors.qo","sync":true,"body":{"Sensor_A_1_1":1191,"Sensor_A_1_2":1168,"Sensor_A_1_3":1249,"Sensor_A_2_1":579,"Sensor_A_2_2":571
{"total":1}

Fig 17. Notehub data representation

Following figure shows the proof of the data which is received at the notehub end.

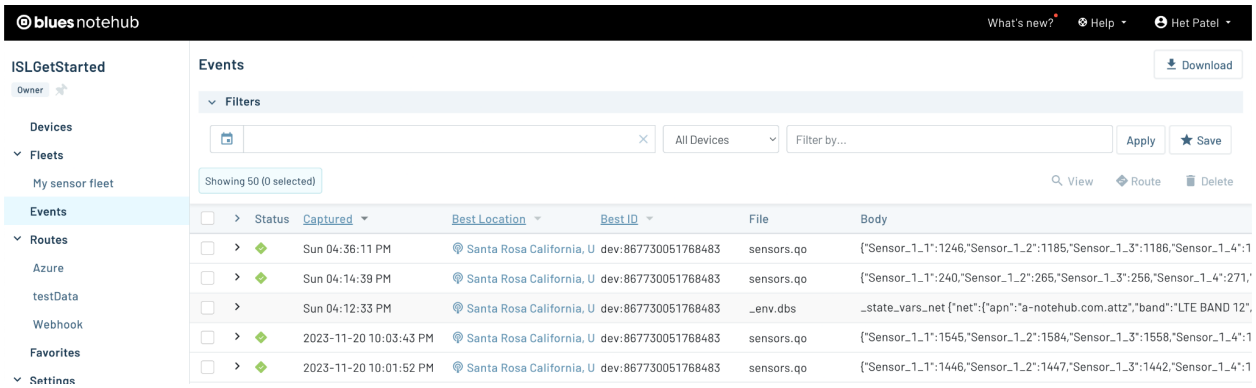

Fig 18. Notehub data representation

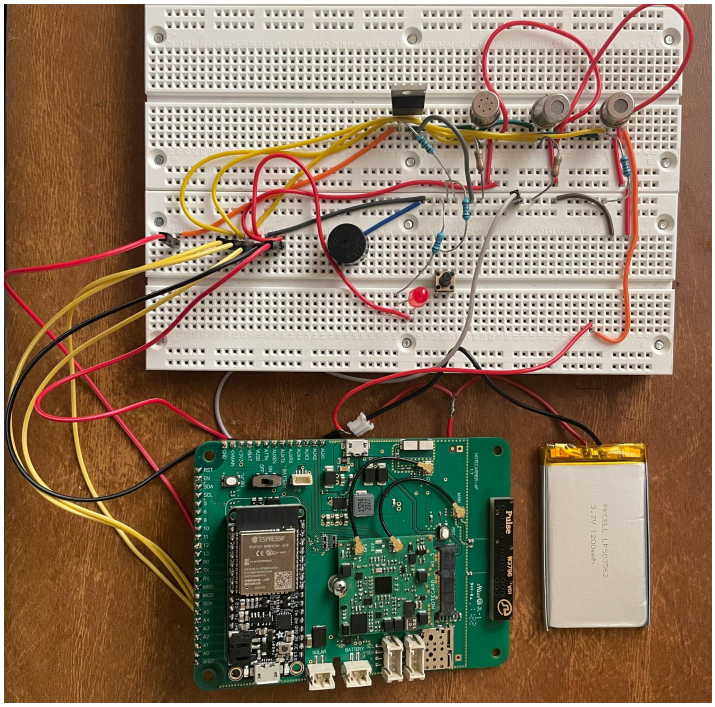

Fig 19. Breadboard representation

Chapter 10: Conclusion

In conclusion, the RTOS based breath data collection device development has taken thorough understanding and taken care of each minor component verification in firmware development. This project not only delved into the complexities of real-time task scheduling, sensor interfacing, and data acquisition but also necessitated multiple iterations of the PCB design to align with the project's objectives.

The firmware successfully considers the device functionality, encompassing sensor warm-up, air sampling, user interaction, and data processing. The integration of FreeRTOS facilitated a systematic and organized execution of tasks, improving the readability of the code for the future developers.

The project highlighted the importance of being adaptable and ready for changes as the project evolved. The collaboration between the firmware and hardware sides resulted in a device that efficiently collects breath data.

As technology continues to advance, the experiences gained from this project will undoubtedly motivate me to think more about the future work to make the device more robust and efficient.

Appendix

BENEFITS OF RTOS

There are several advantages of using an RTOS over a simple Superloop (infinite while loop).

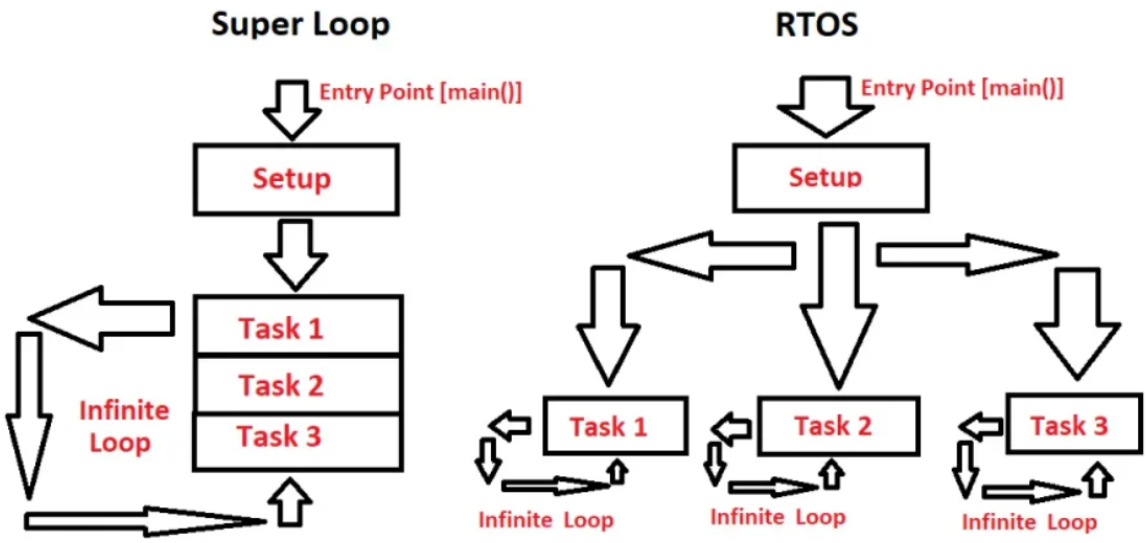

Fig 7. Superloop vs RTOS

WHEN

An RTOS is commonly used when there are more interrupt sources, more functions, and more communication interfaces. Overall, it's the application complexity that mostly decides the use of RTOS tailored for MCUs.

RAM?
As we saw earlier in the code.