



**PAMANTASAN NG LUNGSOD NG MAYNILA**  
(University of the City of Manila)  
Intramuros, Manila

---

---

Embedded Systems Laboratory

Final Project

---

# IoT Air Quality Monitoring System

---

*Submitted by:*

Paz, Kristel Erica D. <Leader>

Bueno, Theron Adrienne A.

Cruz, James Laurence A.

Lenizo, Jackilyn O.

Tuling, Jeanne Rose P.

*Instructor:*

Engr. Maria Rizette H. Sayo

January 14, 2023

---

---

# I. Objectives

## Introduction

In a world where air pollution is a growing concern, the need for accurate and reliable air quality monitoring systems has never been more pressing. A paper by Fenger remarked that problems with air pollution had been known for millennia, but the attitude towards them was ambiguous [1]. Today, the advent of COVID-19 has affected millions of people and the increased concern of the virus spreading in confined spaces due to meteorological factors has sequentially fostered the need to improve indoor air quality [2]. This research project aims to address this need by creating an Internet of Things (IoT) system that monitors an area's air quality.

The system is built on the principles and techniques acquired in the Embedded Systems Laboratory course, utilizing the low-cost and low-power ESP32 microcontroller and the MQ135 air quality sensor. The sensor can detect a wide range of gases in the air, including ammonia, alcohol, benzene, smoke, and carbon dioxide. The sensor is connected to the ESP32, which gathers and sends the data to a live graphical data visualization using web development tools Svelte and Chart.js and a Firebase database.

The system is cost-effective and user-friendly, providing real-time data visualization of the air quality, and making it easier for people to understand the air they breathe. The system also has the capability to store data, enabling users to track changes over time and make informed decisions about their environment.

With the increasing need for air quality monitoring, this IoT-based system offers a solution that is both practical and accessible. It represents a significant step forward in the fight against air pollution and the hope for a cleaner and healthier future.

The objective of this study is to create an IoT based system that monitors air quality of an area. This project aims to implement the principles and techniques acquired in the course Embedded System Laboratory.

Correspondingly, the specific objectives of this research are as follows:

1. To create an IoT-based Air Quality Monitoring System using ESP32 and MQ 135 Air quality sensor.
2. To create a live graphical data visualization of the air quality using Svelte and Chart.js for web development and Firebase as database.

## II. Methods

### **Theories and Principles Behind MQ-135 Gas Sensor**

An MQ135 air quality sensor is a type of MQ gas sensor, which is housed in a steel exoskeleton. This sensor is used to detect harmful gases, and smoke in the fresh air; measure, and monitor a wide range of gases present in air like ammonia, alcohol, benzene, smoke, carbon dioxide, etc. Through connecting leads, this sensing element is exposed to current. The gases that are close to the sensing element are ionized by this current, which is also known as a heating current, and are subsequently absorbed by the sensing element. As a result, the resistance of the sensing element changes, changing the amount of current that flows out of it. Moreover, it is inexpensive and is suitable for air quality monitoring. However, it is not very accurate as it detects several harmful gases but does not tell what exact concentration of the detected gas is present in the air; and, it has needs for calibration [3].

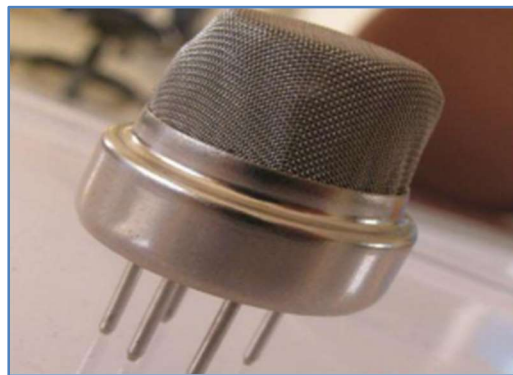


Figure 1 MQ-135 Gas Sensor

The 4-pin sensor module for the MQ135 air quality sensor offers analog and digital output from the corresponding pins. Following is a figure of the pin layout for the MQ135 air quality sensor.

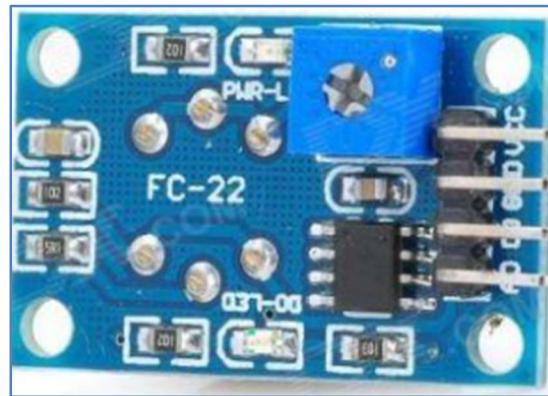


Figure 2 Pin Configuration MQ135 Gas Sensor

**VCC:** The MQ135 sensor module is powered by a positive 5V power supply, which is represented by this pin.

**GND (Ground):** This pin serves as a reference potential and links the ground to the MQ135 sensor module.

**Digital Out (Do):** This pin is a reference to the digital output pin, which produces a digital output by using a potentiometer to change the threshold value. The MQ135 sensor functions without a microcontroller due to this pin, which is used to detect and measure any specific gas.

**Pin 4: Analog Out (Ao):** This pin produces an analog output signal that ranges from 0 to 5 volts and is dependent on the gas pressure. The MQ135 sensor module measures the concentration of gas vapor, and this analog output signal is proportional to that value. The gases are measured in PPM using this pin. It is mostly interfaced with microcontrollers, operates on 5V, and is driven by TTL logic.

## ESP32-Cam

ESP32 is a series of low-cost, low-power systems on chip microcontrollers featuring built-in Wi-Fi and dual-mode Bluetooth. The ESP32-CAM contains a very competitive small-size camera module with a footprint of only 27\*40.5\*4.5mm and a deep sleep current of up to 6mA that can run independently as a minimum system. It is a fully functional microcontroller with a built-in video camera and microSD card slot that can be useful to store images taken with the camera or to store files to serve to clients. It is appropriate for home smart devices, industrial wireless control, wireless monitoring, QR wireless identification, wireless positioning system signals and other IoT applications. It is the perfect solution for IoT applications because it is affordable, simple to use, and appropriate for IoT devices that need a camera with advanced features like image tracking and identification [4].

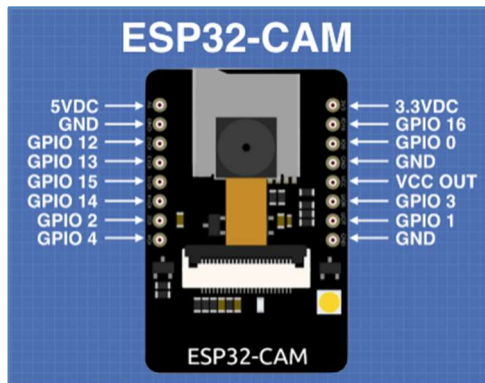


Figure 3 Pinout of the ESP32-Cam Module

A subset of the ESP-32 module called ESP32-Cam has pinouts that include an external antenna and a white LED for flashing. It can be powered by any of the two available voltages—3.3 or 5 volts direct current—but not both. That voltage will be reflected in the VCC output. Additionally, the modules transmit and receive functions are handled by the GPIO 1 and 3, respectively. Additionally, the ESP32-Cam module missing an USB port, has a MicroSD card with a maximum storage capacity of 4GB, and only supports 4GB on larger microSD cards.

## Carbon Dioxide

The fourth most common gas in the atmosphere is carbon dioxide, sometimes referred to as dry ice, CO<sub>2</sub>, and a diesel exhaust component. Carbon dioxide (CO<sub>2</sub>) is a gas that is colorless, odorless, and inflammable when it is at room temperature. However, depending on the temperature and pressure, CO<sub>2</sub> can also be a liquid or a solid. Because it slowly transforms from a cold solid into a gas, solid carbon dioxide is also referred to as "dry ice". Moreover, carbon dioxide is a byproduct of normal cell function when it is breathed out of the body, as well as when fossil fuels are burned or decaying vegetation. Carbon dioxide is used as dry ice in stage and theatrical shows to create fog, as well as in laboratories, fire extinguishers, and firefighting equipment. If the air is not vented, the usage of dry ice might cause indoor CO<sub>2</sub> levels to rise [5].

Additionally, the gas can infiltrate into basements where there are high soil CO<sub>2</sub> levels through stone walls or fractures in flooring and foundations. Buildings that house a lot of people or animals may also have CO<sub>2</sub> buildup, which is a sign that there are issues with the building or home's fresh air circulation. High CO<sub>2</sub> levels have the potential to be harmful to human health because they can displace oxygen and nitrogen. How much fresh air is introduced into a facility typically affects how much carbon dioxide is present there. In general, the amount of fresh air exchange decreases as the building's CO<sub>2</sub> level rises. As a result, checking the CO<sub>2</sub> levels in interior air can show whether the HVAC systems are performing as intended. CO<sub>2</sub> concentrations are often expressed as a percentage of air or parts per million (ppm). High carbon dioxide levels can cause poor air quality, and high CO<sub>2</sub> levels, often over 1000 ppm, signal a potential issue with fresh air circulation in the area. CO<sub>2</sub> levels typically signal that the HVAC system needs to be looked at.

The levels of CO<sub>2</sub> in the air and potential health problems are: 400 ppm is the average outdoor air level; 400–1,000 ppm is the typical level found in occupied spaces with good air exchange; 1,000–2,000 ppm is the level associated with complaints of drowsiness and poor air; 2,000–5,000 ppm is the level associated with headaches, sleepiness, and stagnant, stale, stuffy air. Poor concentration, loss of attention, increased heart rate and slight nausea may also be present; 5,000 ppm is the level that indicates unusual air conditions where high levels of other gases could also be present. Toxicity or oxygen deprivation could occur. This is the permissible exposure limit

for daily workplace exposures; and 40,000 ppm is the level that is immediately harmful due to oxygen deprivation. As a result, fresh air exchange decreases as CO<sub>2</sub> levels rise.

## Implementation Hardware

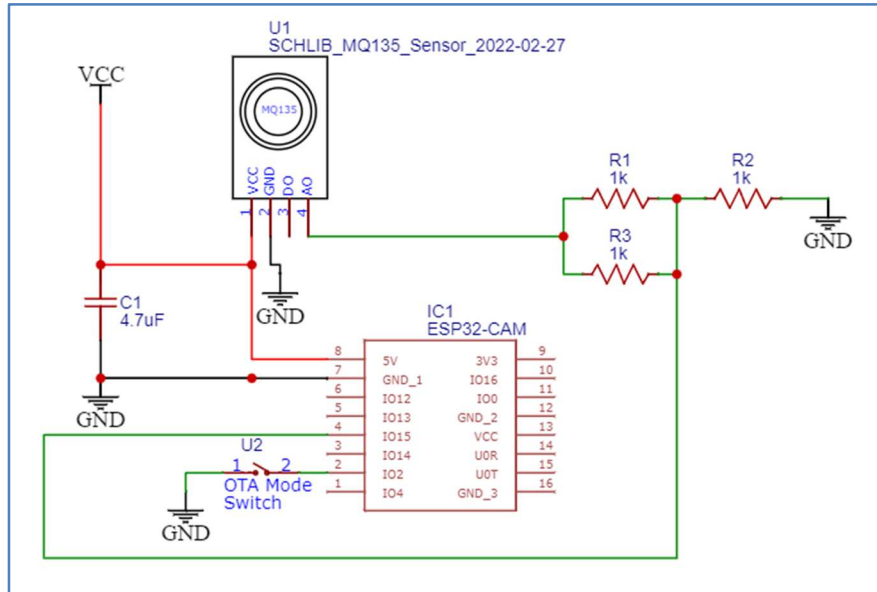


Figure 4 Iot Air Quality Monitoring System Circuit Diagram

The image shows the schematic diagram of the system's hardware circuit. The main components used were an ESP32-CAM used as a microcontroller, and an MQ135 air quality sensor. The ESP32-CAM was chosen for convenience since it is already on-hand, however, other Arduino-based or compatible microcontrollers can also be used. The main power rails used in powering the sensor and microcontroller is 5V, however, since the microcontroller is sensitive to noisy power sources, a 4.7uF capacitor is placed between 5V and ground to smoothen out ripples coming from the power supply. The analog output of the gas sensor ranges from 0V-5V, but this cannot be directly connected to the ESP32 since the microcontroller operates at a 3.3V TTL level. Due to this, a voltage divider regulates the analog output of the gas sensor to a manageable range of 0V-3.3V. This is done by the resistors R1, R2, and R3. Ideally, the voltage divider should have values of 1kΩ and 500Ω, but to save materials costs by buying bulk single value resistor packs, an equivalent circuit was constructed using three (3) 1kΩ resistors. The switch U2 is connected to a

digital I/O pin and will be used to control the mode of the microcontroller as outlined in a later section.

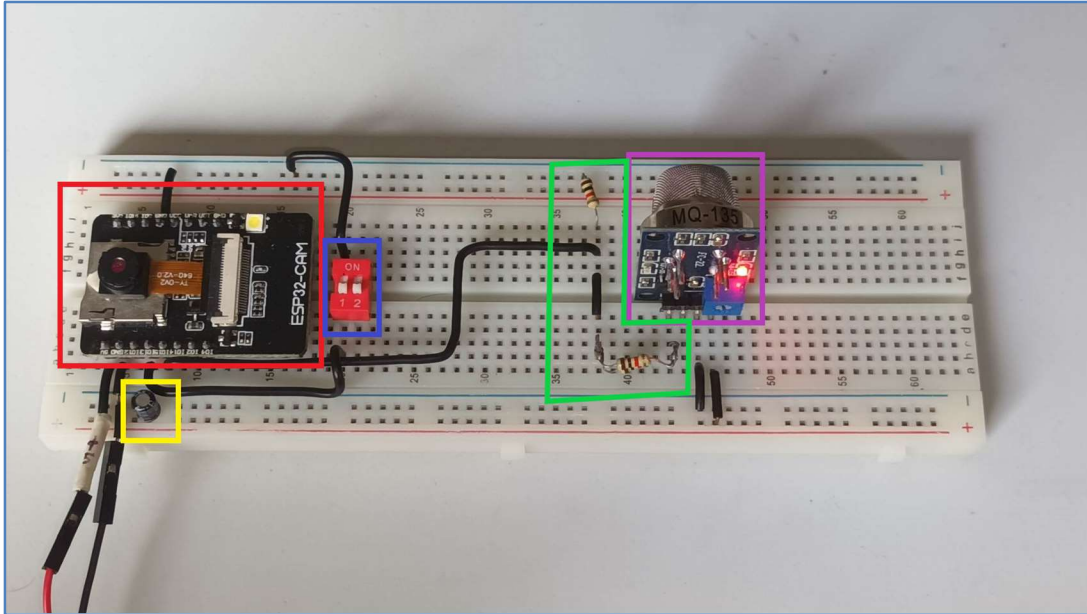


Figure 5 Physical Layout

The prototype was constructed on a breadboard for ease of development. The ESP32-CAM microcontroller board is outlined in red. Directly below it are the power supply input and the 4.7uF bypass capacitor outlined in yellow. Beside the microcontroller lies the OTA mode switch. On the right side of the breadboard in the image, the MQ-135 gas sensor and the resistor divider regulator can be found. The gas sensor is outlined in violet, while the resistor divider network is outlined in green.

### Sensor Calibration

The MQ135 Arduino library [6] was used in calibrating and reading parts-per-million (PPM) values from the MQ-135 sensor. Some modifications were made to the library in order to make baseline values relevant in at time of writing, and to also adjust to differences in hardware obtained by the researchers. The changes are as follows:

1. Change the formula of converting the analog value to the parts-per-million (PPM) value to reflect the 12-bit resolution of the ESP32 analog-to-digital converter (ADC).



$$\text{PPM} = \left( \frac{1023}{V_{in}} * 5V - 1 \right) * R_L$$

to

$$\text{PPM} = \left( \frac{4095}{V_{in}} * 3.3V - 1 \right) * R_L$$

2. Change the baseline atmospheric PPM value of carbon dioxide (CO2) from 397.13 ppm to 414.72 ppm [7].
3. Change the constant value RLOAD from 10.0 to 1.0 in order to reflect the hardware differences of the commonplace MQ135 and the researcher's MQ135.

After the above modifications were made, the researchers have proceeded to calibrating the sensor. First, the MQ135 must be pre-heated for at least 24 hours. After 24 hours of being plugged in, the researchers implemented a function in the ESP32 sketch that calls the library's *getRZero()* function, and outputs the result to the web serial monitor.

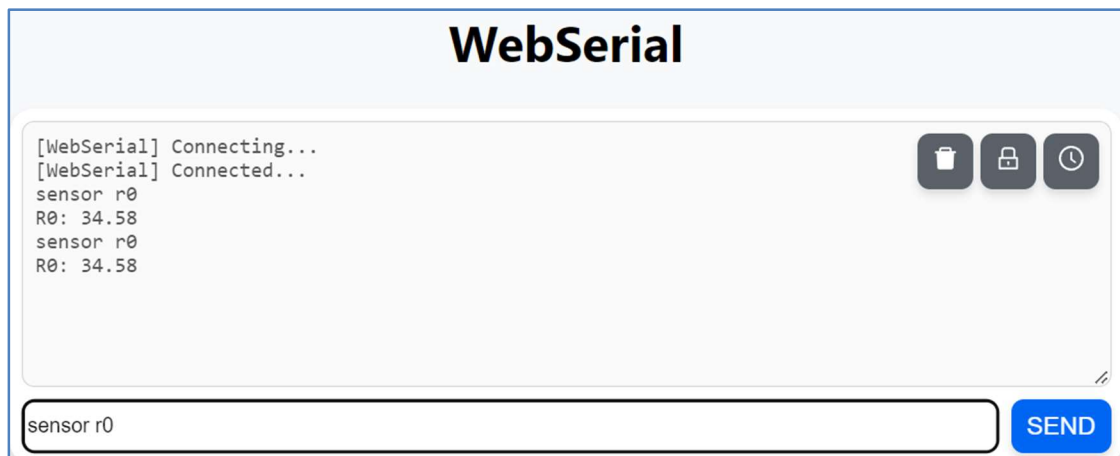


Figure 6 Obtain Sensor R0 for Calibration Using WebSerial

Once the R0 value has been obtained, the header file for the MQ135 library needs to be modified again in order for the library to use the new calibration values.

## System Flowchart

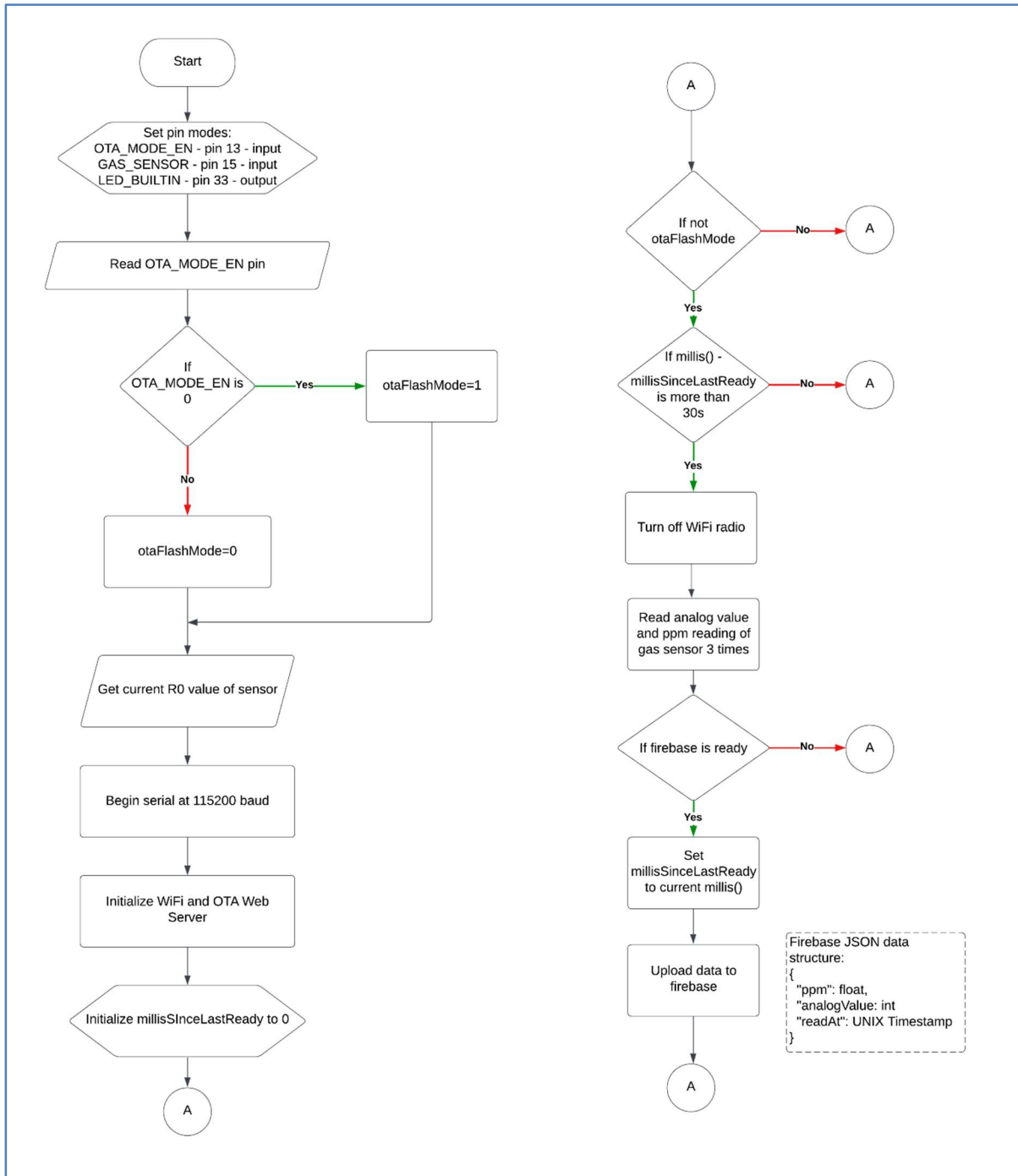


Figure 7 Flowchart

To begin, the researchers must first define at the start of the program the pin modes utilized for this project. Pin mode 13 input was used to set the OTA mode, pin mode 15 input was used to

set the gas sensor and lastly the pin mode 33 output was used for LED built in. Then it will read the OTA flashmode. If OTA flashmode is ON, only OTA will work, everything will be disabled. Also, no data will be uploaded to the Firebase but the Wi-Fi is ON. This ensures that we can reupload code to the device even if we upload broken firmware to it. If OTA flashmode is OFF, OTA and everything else will still work, but OTA might not be fast or reliable.

## **PlatformIO IDE**

For ESP32 development, the researchers used PlatformIO instead of Arduino IDE because of its features that helps in creating and troubleshooting the code much easier. Arduino is an open-source prototyping platform for electronics built on flexible, user-friendly hardware and software. On the other side, PlatformIO is described as the "Next generation IDE for IoT".

To develop embedded applications, embedded system and software engineers can use PlatformIO, a professional IDE tool that works across platforms, architectures, and frameworks. PlatformIO provides a common IDE interface, allowing you to program your hardware in a more developer-friendly manner. As plugins, PlatformIO is compatible with a number of the most widely used integrated development environments (IDEs) and text editors [8].



Figure 8 Development Environment Used

All of the members of the researchers have Visual Studio Code on their computers, so they simply added the PlatformIO IDE by clicking the extension button on the left side of VSCode and installing the plugin.

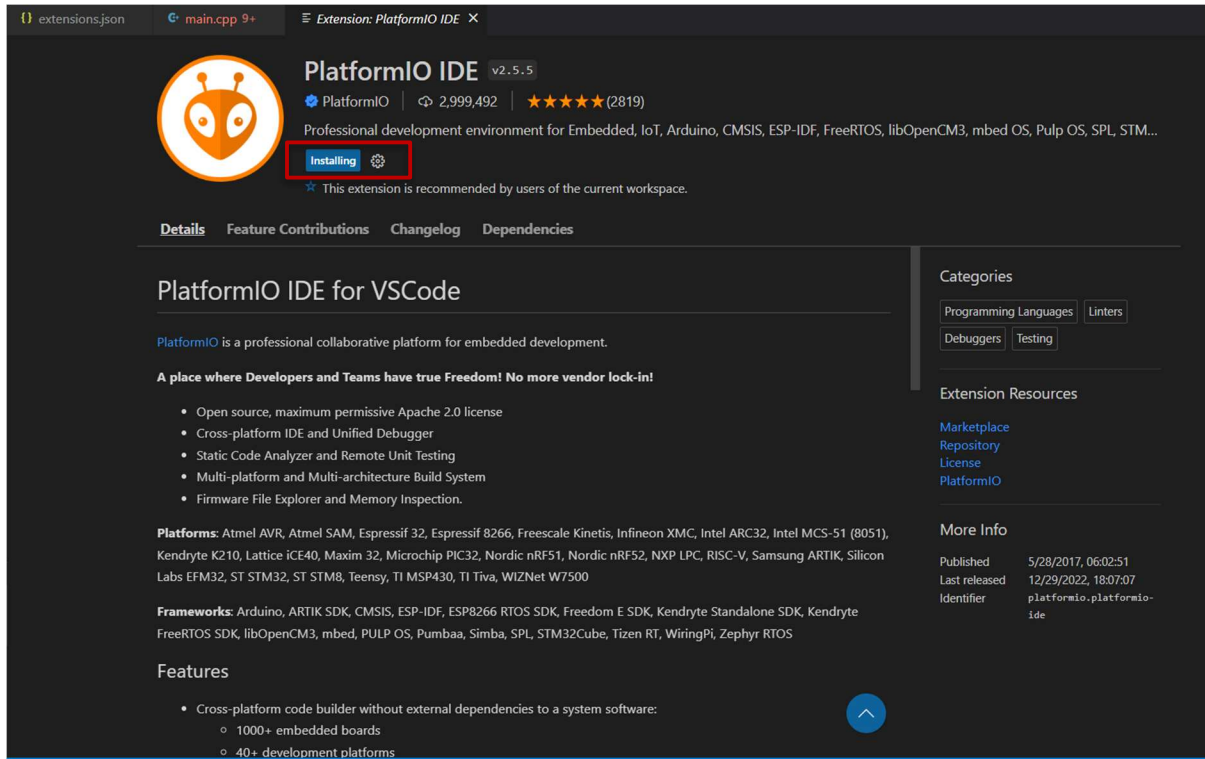


Figure 9 Installation of PlatformIO IDE

Along with installing the PlatformIO plugin, it will also add Visual Studio Code extensions that will enable it to comprehend C and C++ code. Once finished, the researchers configure PlatformIO to support the ESP32 since by default it doesn't.

After the installation has been completed, a new PlatformIO icon has been added to Visual Studio Code. Clicking it will open the PlatformIO home page under quick access. In this view, the team selected Platform, then Embedded, and then searched for Espressif 32 which is the manufacturer of the ESP32.

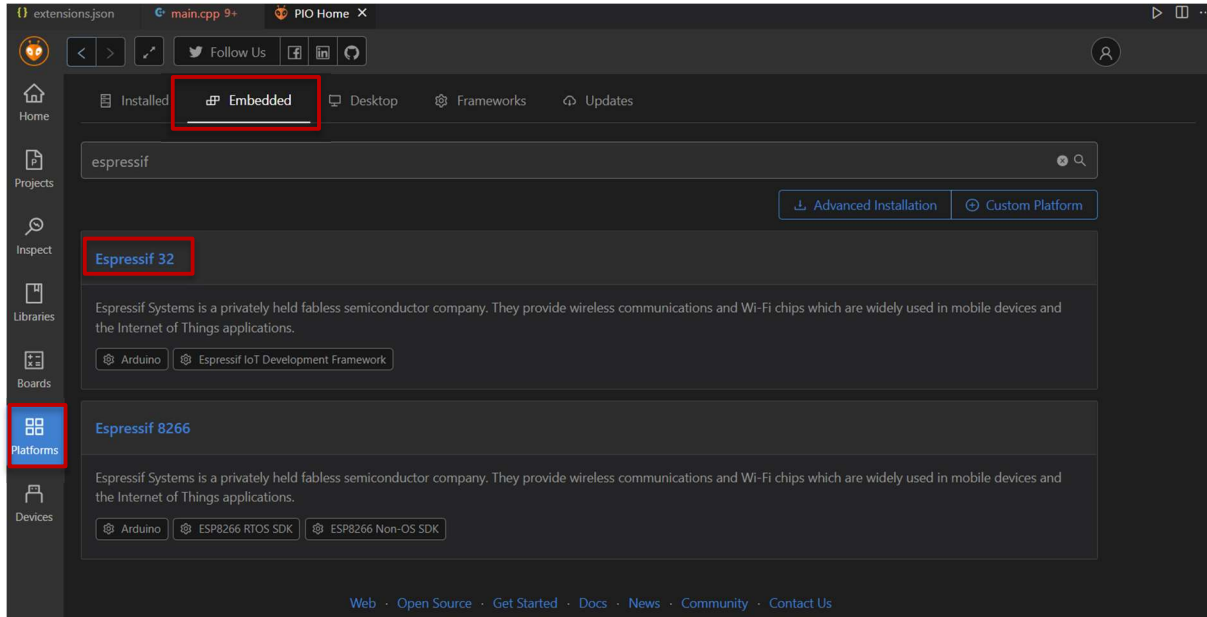


Figure 10 Searching for Espressif 32

The researchers selected the install button to add support for the ESP32 which will download and install the software development kit for the ESP32. Once done, the researchers can now create their project in PlatformIO.

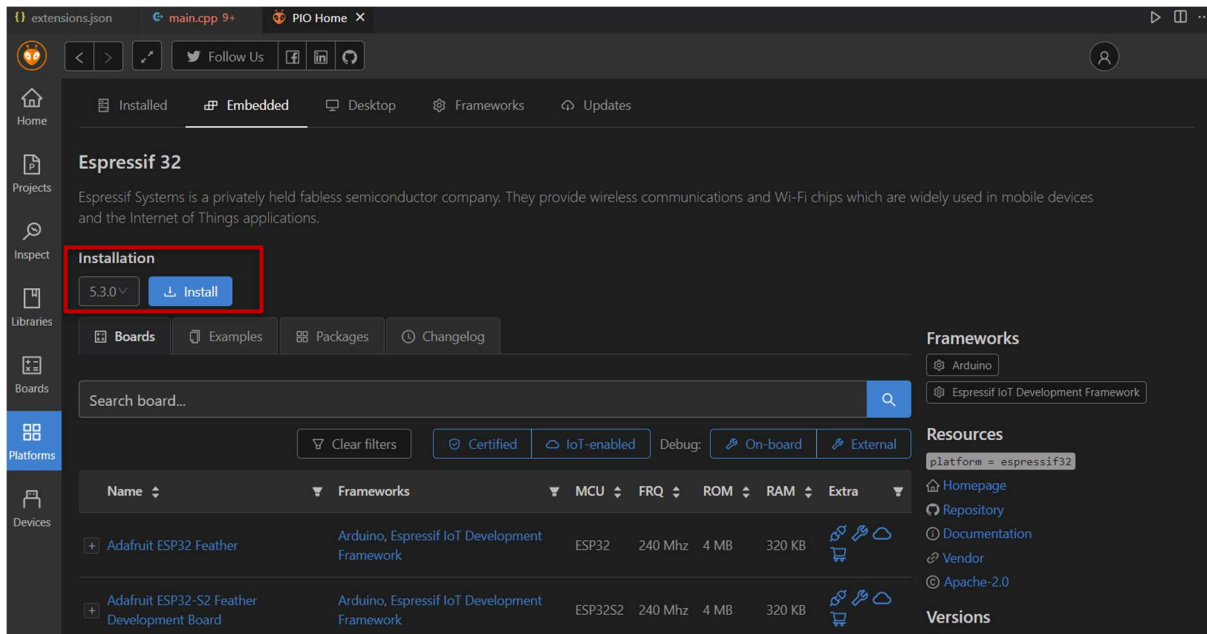
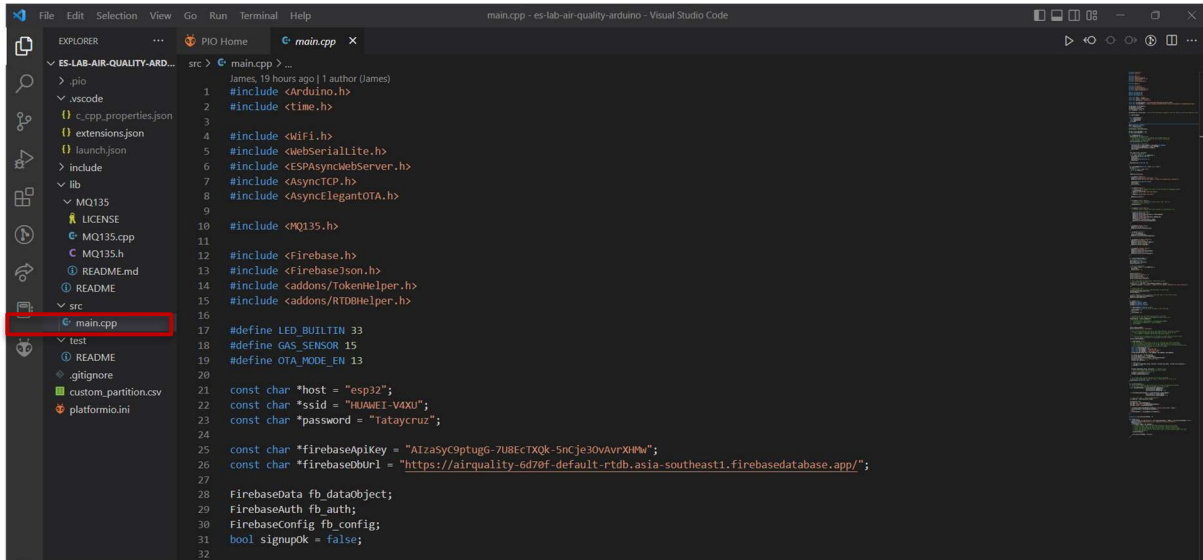


Figure 11 Installation of Espressif 32

The main source code of the project can be found in the source code directory and under that there is a file called main.cpp. In this location, the team writes all of the source code for the ESP32 and includes libraries that are compatible with it.



```
main.cpp - es-lab-air-quality-arduino - Visual Studio Code
EXPLORER
  ES-LAB-AIR-QUALITY-ARD...
    src
      main.cpp
    pio
    .vscode
    c_cpp_properties.json
    extensions.json
    launch.json
    include
    lib
    MQ135
    LICENSE
    MQ135.cpp
    MQ135.h
    README.md
    README
    src
      main.cpp
    test
    README
    .gitignore
    custom_partition.csv
    platformio.ini

main.cpp
1 James, 19 hours ago | 1 author (James)
2 #include <Arduino.h>
3 #include <time.h>
4 #include <WiFi.h>
5 #include <WebSerialLite.h>
6 #include <ESPAsyncWebServer.h>
7 #include <AsyncTCP.h>
8 #include <AsyncElegantOTA.h>
9
10 #include <MQ135.h>
11
12 #include <Firebase.h>
13 #include <FirebaseJson.h>
14 #include <caddons/TokenHelper.h>
15 #include <caddons/RTDBHelper.h>
16
17 #define LED_BUILTIN 33
18 #define GAS_SENSOR 15
19 #define OTA_MODE_EN 13
20
21 const char *host = "esp32";
22 const char *ssid = "HUMIEI-V4XU";
23 const char *password = "Tataycruz";
24
25 const char *firebaseApiKey = "AIzaSYC9ptugg-7UBECTXQk-5nCje30vAvrXI4W";
26 const char *firebaseDbUrl = "https://airquality-6d79f-default-rtdb.asia-southeast1.firebaseio.com/";
27
28 FirebaseData fb_dataObject;
29 FirebaseAuth fb_auth;
30 FirebaseConfig fb_config;
31 bool signupok = false;
32
```

Figure 12 main.cpp File

Another significant file is called platformio.ini, and it provides the configuration of the system that has been constructed, including the platform, the board, and the framework that the researchers has been utilizing. It also includes the project's dependencies. The first four dependencies were used for debugging since the ESP32 used by the group has no USB connections. The last two dependencies were used for the IoT Air Quality Monitoring System to work.

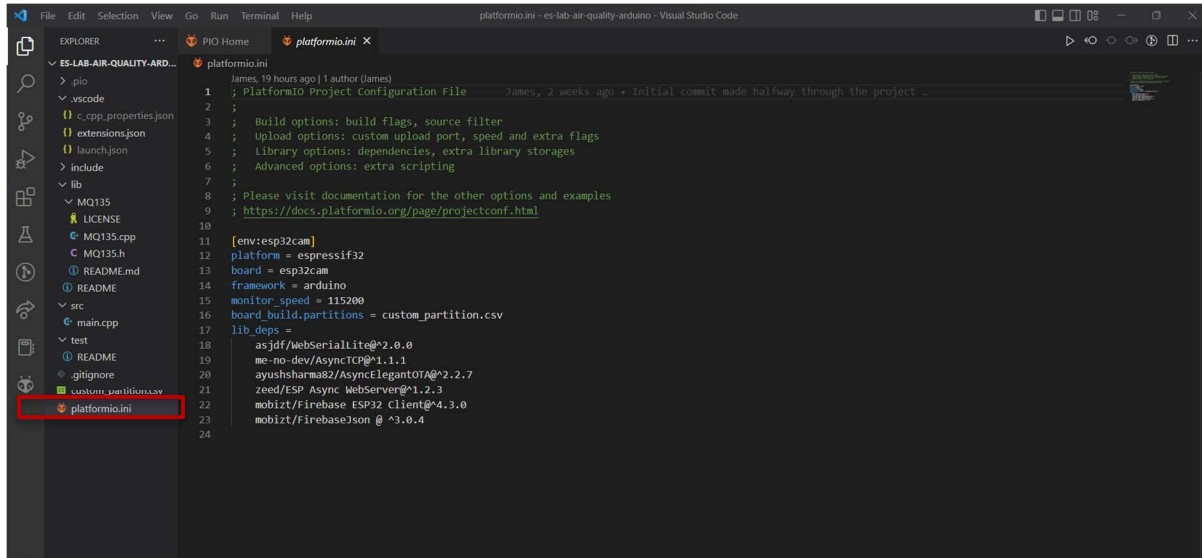


Figure 13 platformio.ini File

The program can be compiled using the button at the bottom left of the IDE called “PlatformIO:Build”.

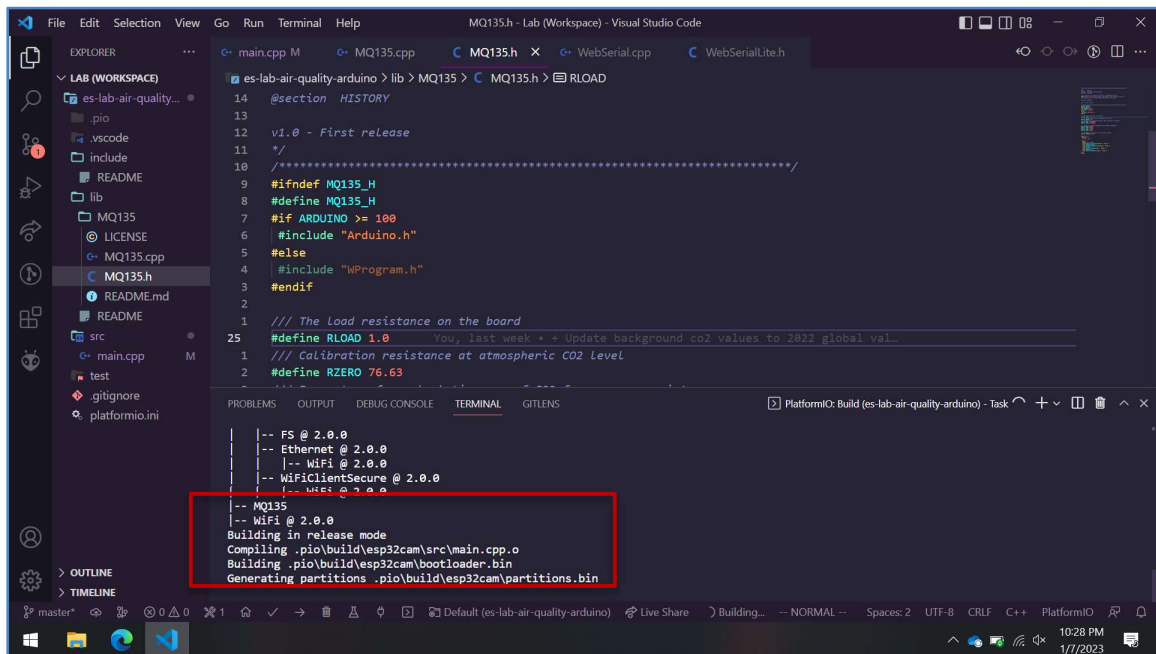


Figure 14 Compiling using PlatformIO in Visual Studio Code

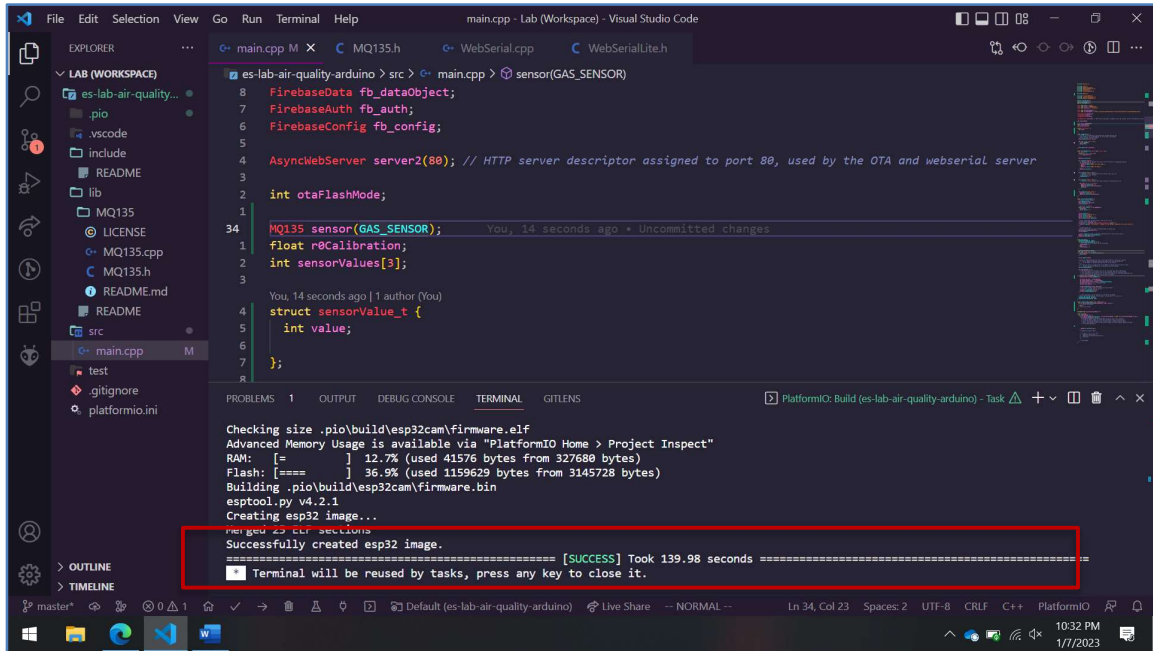


Figure 15 Compiling Completed

## AsyncElegantOTA

In order to program the ESP32, the team utilized Over-the-air Updates (OTA). Through OTA programming, the ESP32 can be updated or uploaded with a new program over Wi-Fi without a USB connection to a computer. It also shortens the time needed for each ESP module to be updated during maintenance. One significant benefit of OTA is the ability to transmit an update to numerous ESPs connected to the same network from a single central point [9].

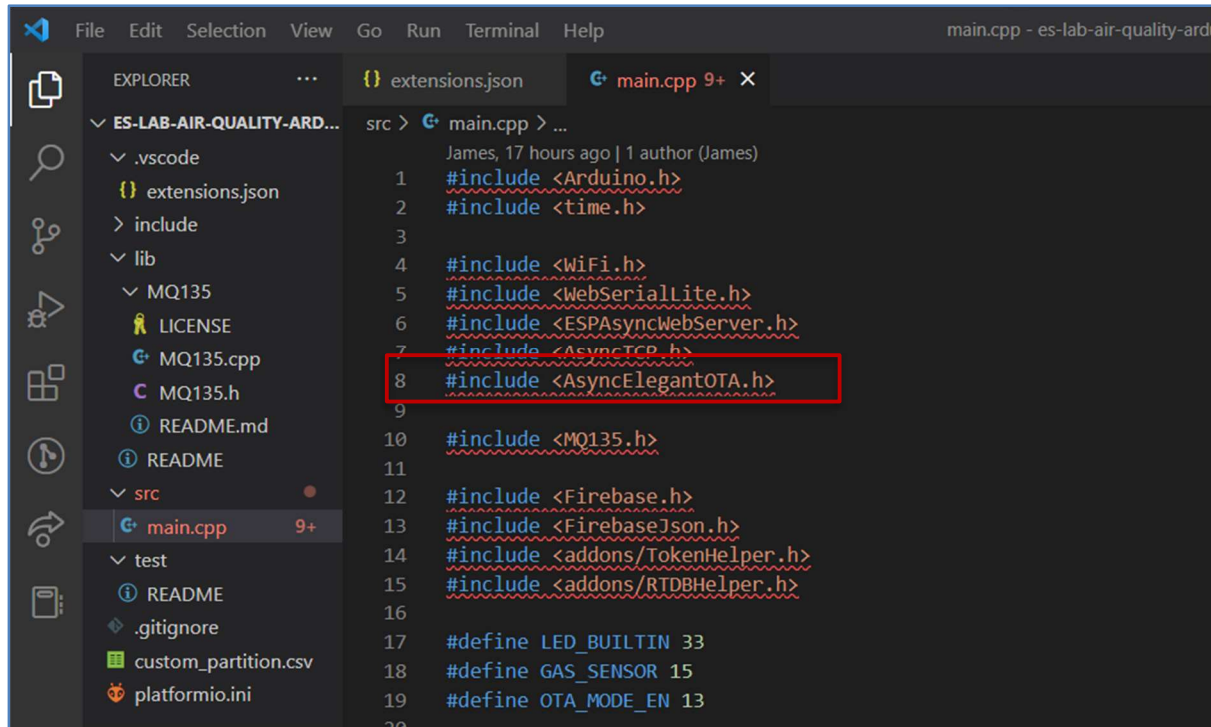
The team utilized ElegantOTA library by Ayush Sharma to perform OTA and to utilize an elegant webpage instead of the typical upload page that comes with the Arduino library [10]. It offers an attractive user interface for uploading Over-the-Air .bin updates to the ESP Modules. The precise status and progress of the upload are presented on the user interface.



Figure 16 Library Used



To utilize the ElegantOTA, the researchers simply include the library on top of the ESP32 Code.



```
File Edit Selection View Go Run Terminal Help main.cpp - es-lab-air-quality-ard...
EXPLORER
ES-LAB-AIR-QUALITY-ARD... src > main.cpp > ...
  .vscode
  extensions.json
  include
  lib
  MQ135
  LICENSE
  MQ135.cpp
  MQ135.h
  README.md
  README
  src
  main.cpp 9+
  test
  README
  .gitignore
  custom_partition.csv
  platformio.ini
1 James, 17 hours ago | 1 author (James)
2 #include <Arduino.h>
3 #include <time.h>
4 #include <WiFi.h>
5 #include <WebSerialLite.h>
6 #include <ESPAsyncWebServer.h>
7 #include <AsyncTCP.h>
8 #include <AsyncElegantOTA.h>
9
10 #include <MQ135.h>
11
12 #include <Firebase.h>
13 #include <FirebaseJson.h>
14 #include <addons/TokenHelper.h>
15 #include <addons/RTDBHelper.h>
16
17 #define LED_BUILTIN 33
18 #define GAS_SENSOR 15
19 #define OTA_MODE_EN 13
20
```

Figure 17 ElegantOTA Library

To add new firmware or files to the filesystem, the user can use the web server that the ElegantOTA library generates and accesses through their local network (SPIFFS). The uploaded files must be in .bin format [11].

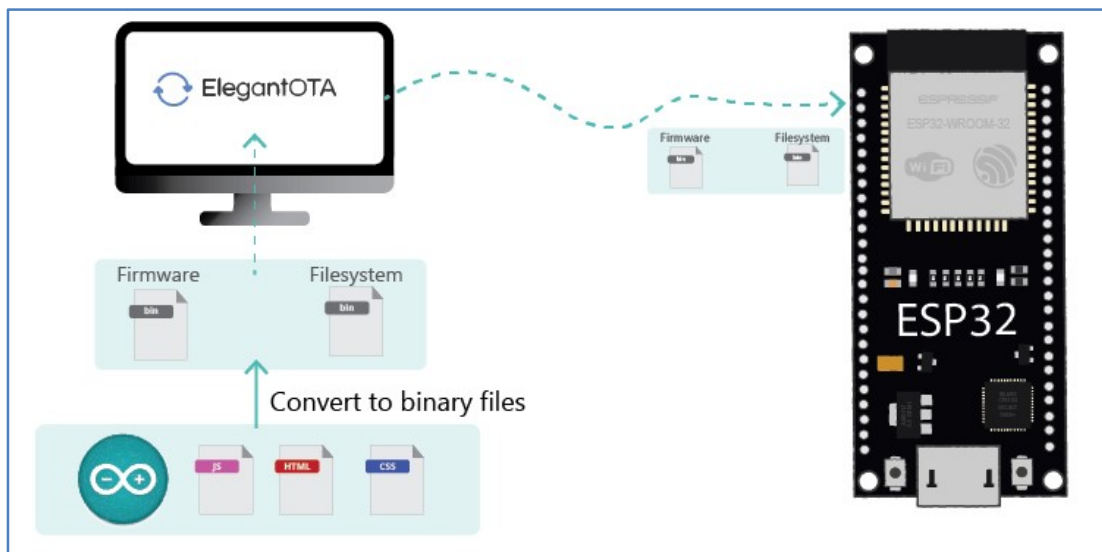


Figure 18 Over-The-Air Updates using ESP32 Board [11]

There are only three primary steps involved in the implementation of basic OTA, which makes it an extremely straightforward process. To begin, everyone in the researchers downloaded the most recent version of Python and installed it on their individual computers. The OTA firmware is then loaded onto the ESP32 using a serial interface by the team. To enable the ESP32 to connect to an existing network, the researchers must alter the following two variables with their network credentials before uploading the sketch:

```
const char* ssid = ".....";  
const char* password = ".....";
```

This must be done before any further over-the-air upgrades may be implemented. After completing this procedure, the team will be able to send new sketches to the ESP32 over-the-air. Because the ESP32 includes a Wi-Fi, the team is able to use the ESP32 module's Wi-Fi connectivity to transfer a software to the device. The URL includes a local IP address along with "/update" to access the OTA. This is how the URL will appear: http://<IPAddress>/update

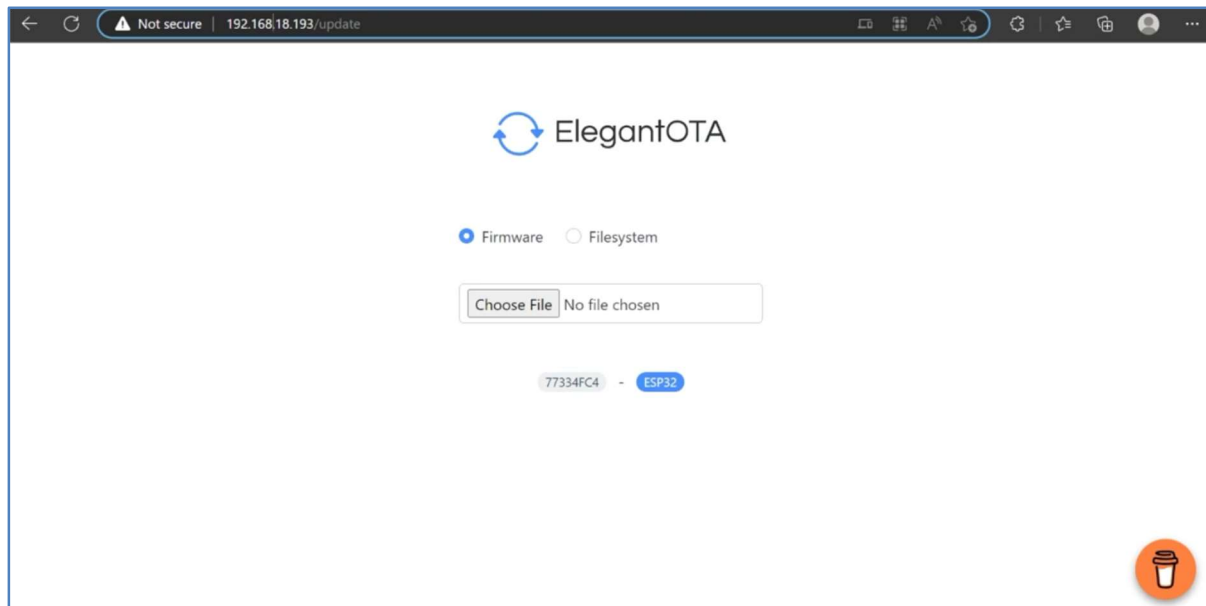


Figure 19 ElegantOTA Web Interface

To upload new firmware, the group clicked the “Choose File” button located at the web interface. The group uploads the firmware.bin file and starts the updating process which will take a few seconds, then the newly updated program will run in the ESP32.

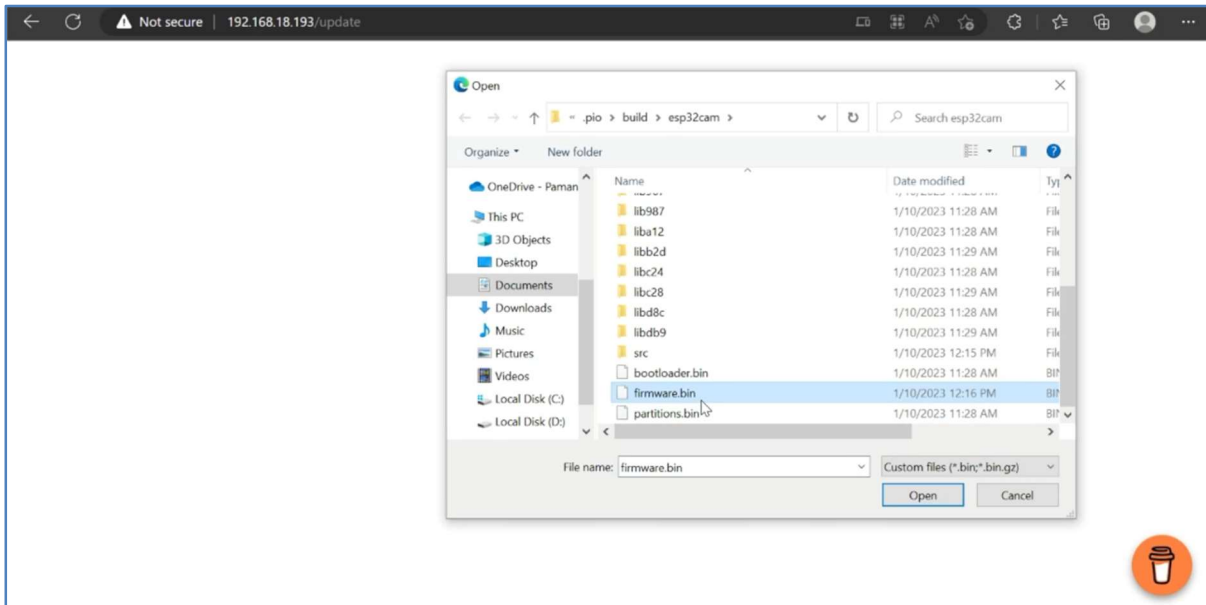


Figure 20 Uploading New Firmware using ElegantOTA

This library displays the current upload progress of your OTA and, once that process is complete, it will reveal the status of your OTA.

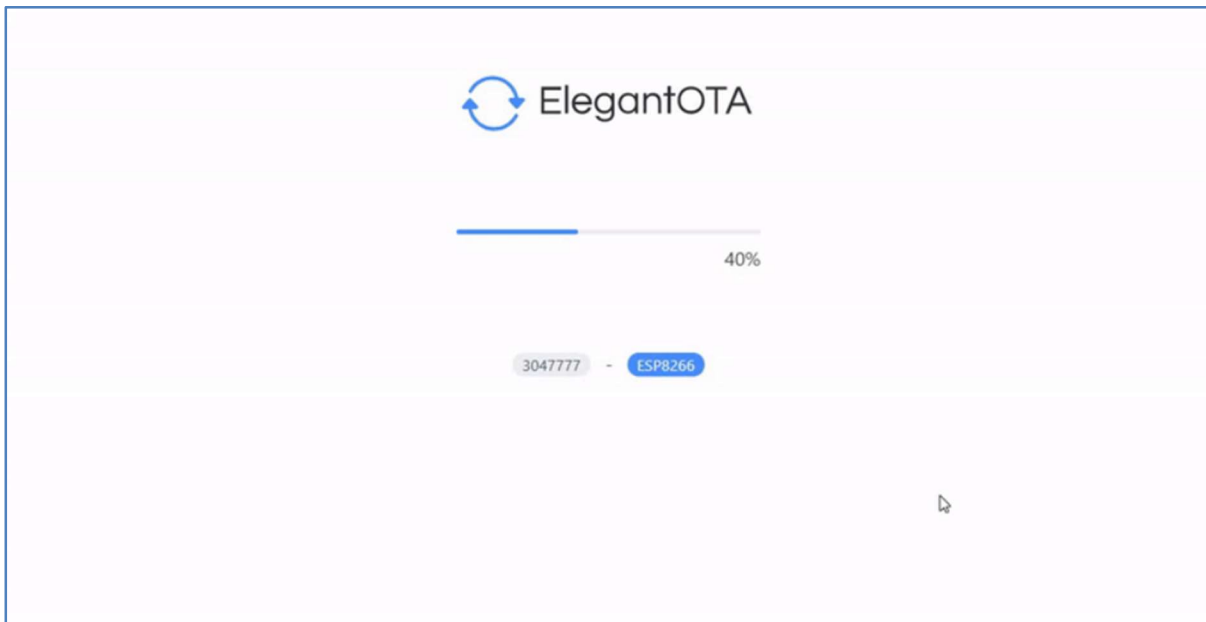


Figure 21 Process of Uploading New Firmware

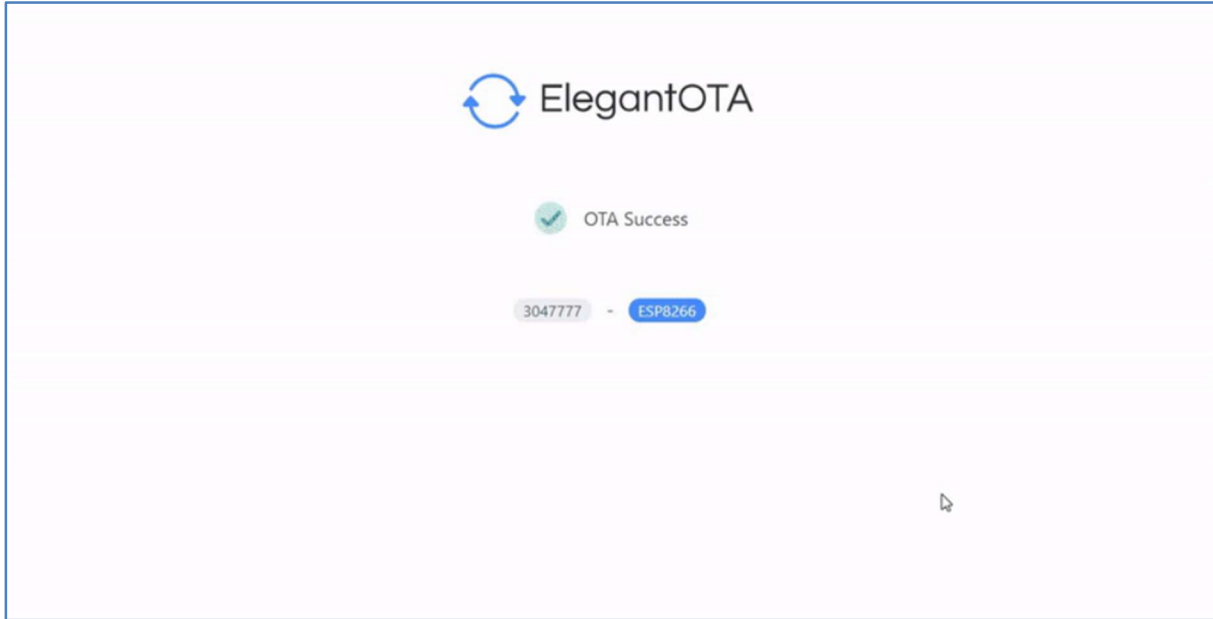


Figure 22 Uploading Success

## Firestore Realtime Database

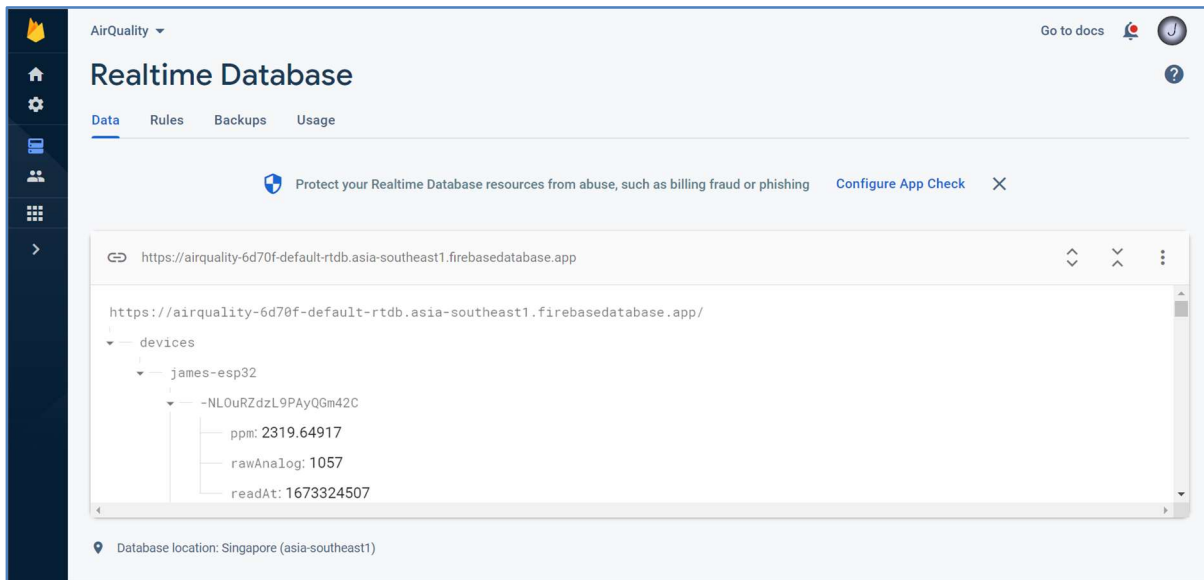


Figure 23 Firestore Realtime Database (RTDB)

The ESP32 uploads data to an instance of a Firestore RTDB every 30 seconds. The data has a simple structure consisting of a JavaScript Object Notation (JSON) object with the parameters *ppm*, *rawAnalog*, and *readAt*. The *rawAnalog* value was not used in the frontend web application, however it has been kept in place by the researchers as a contingency.

## **Svelte Frontend Web Application**

The system's graphical user interface was developed using Svelte. Svelte is a modern, open-source JavaScript framework for web application development. With its features, it offers developers a unique experience. Svelte requires less code, which saves time, minimizes errors, and makes the code simpler to read. Because it is a compiler, it generates code that directly manipulates the DOM rather than a library that requires extensive abstraction layers. Furthermore, it is truly reactive, allowing its customers to design applications according to their needs without worrying about excessive overhead.

Components in Svelte are developed using a template syntax similar to that employed by other JavaScript frameworks. A svelte component can be constructed with HTML, CSS, and JavaScript, which are all utilized in the development of our website. The HTML code establishes the component's structure and layout, the CSS code determines its styling, and the JavaScript code describes its behavior [12], [13].

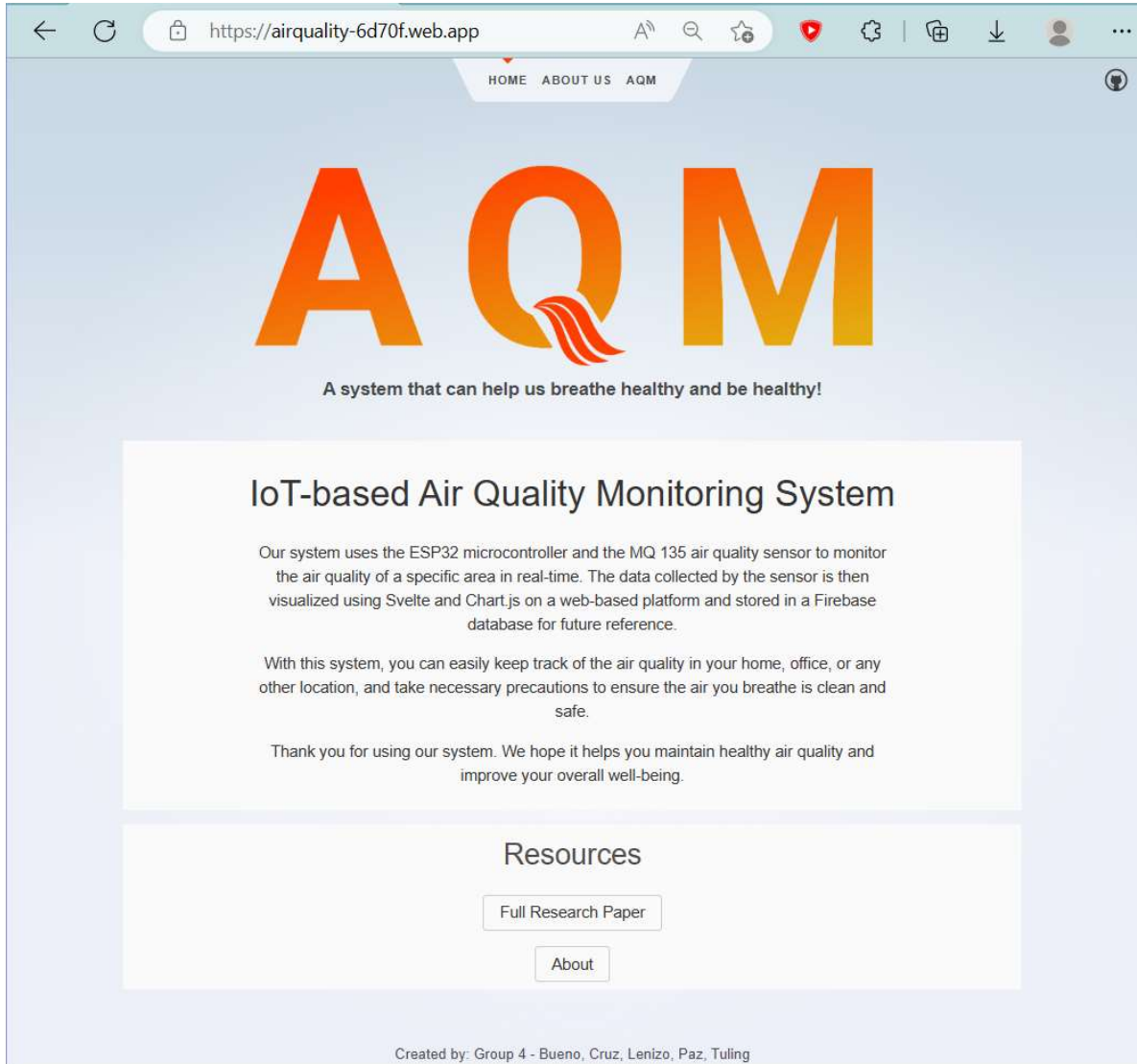


Figure 24 Home page of the Website

## Welcome to our Website!

We started as a group of college individuals who share the same interest about air quality in the Philippines by distributing more information for those individuals who are concerned about the quality of air in their area. That is why the group has created an IoT-based Air Quality Monitoring System powered by ESP32 Microcontroller and MQ135 Air Quality Sensor which displays live air quality where the hardware is located. The air quality monitoring system is a critical piece of information on your home or office environment. It accurately measures the quality of indoor air, which can help you find sources of unhealthy pollutants and their effects on your health.

Air quality has become a significant global concern for the human health and welfare. Too often, air pollution is an invisible killer we don't notice until it's causing various diseases such as asthma, COPD and heart disease. In addition to other symptoms caused by the inhalation of particulate matter, people's daily lives were influenced by noise levels from traffic, shipping traffic or ventilation systems.

Along the way, we aspire to improve our system by providing insights to users on how to improve the natural ventilation in their homes or office, as well as recommend solutions for better indoor air quality control.

## Meet Our Team



Theron Adrienne A. Bueno  
2019-10752



James Laurence A. Cruz  
2019-02106



Jackilyn O. Lenizo  
2019-02194



Kristel Erica D. Paz  
2019-02198



Jeanne Rose P. Tuling  
2019-20396

Figure 25 About Us page of the Website

Our GUI consists of three tabs: "Home," "About Us," and "AQM," respectively. Our system's brief description and a link to our manuscript are the key features of the homepage. The following tab is the "about" page, which details everything about the team. Finally, which is the primary focus of the GUI, the AQM or the Air Quality Monitoring tab shows a graphic representation of the data our system has obtained.



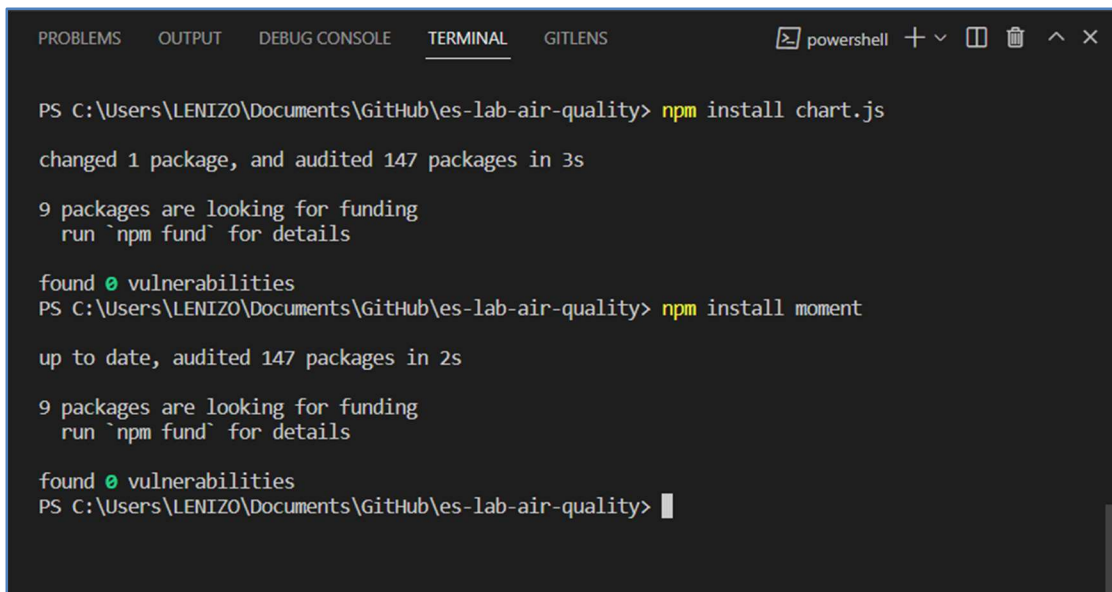
Figure 26 Data visualization using Svelte Libraries

Chart.js is one of the Svelte libraries that we have installed on our system. It is a community-maintained open-source JavaScript charting library that is MIT licensed. It provides support for eight different responsive chart types, all of which are highly customizable. It is particularly useful for easily visualizing data because it offers developers plugins and customization options to add chart functions such as annotations, zoom, or drag-and-drop, to mention a few. Chart.js performs exceptionally well when applied to very large datasets. These types of datasets can be efficiently absorbed by utilizing the internal format, which enables the data parsing and normalization processes to be skipped [14]. Furthermore, we have maximized chart.js by utilizing chart options. Developers can add plugin annotations to the chart area to enhance data visualization, such as lines, boxes, labels, and several others.

One feature of our data visualization is the ability to view the chart at a variety of dates and times. This was made possible with an additional library named Moment. Moment.js is a



JavaScript library that simplifies parsing, validating, manipulating, and displaying date and time in JavaScript. It enables developers to present human-readable dates based on the user's current location. Moments supports multiple straightforward methods for adding, subtracting, validating, and retrieving the maximum and minimum dates. It's an open-source project, thus developers can easily contribute and add functionality through plugins. Additionally, it can validate dates and parse them into the desired format [15].



```
PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL GITLENS powershell + - [ ] [ ] ^ X
PS C:\Users\LENIZO\Documents\GitHub\es-lab-air-quality> npm install chart.js
changed 1 package, and audited 147 packages in 3s
9 packages are looking for funding
  run `npm fund` for details
found 0 vulnerabilities
PS C:\Users\LENIZO\Documents\GitHub\es-lab-air-quality> npm install moment
up to date, audited 147 packages in 2s
9 packages are looking for funding
  run `npm fund` for details
found 0 vulnerabilities
PS C:\Users\LENIZO\Documents\GitHub\es-lab-air-quality> |
```

Figure 27 Installation of Libraries

```
import moment from 'moment';
import 'chartjs-adapter-moment';
import { getHourlyData, getDailyData } from './chartActions'
import annotationPlugin from 'chartjs-plugin-annotation';
import {
  Chart as ChartJS,
  Filler,
  Title,
  Tooltip,
  Legend,
  LineElement,
  LinearScale,
  PointElement,
  TimeScale,
  TimeSeriesScale,
  LineController,
  Decimation
} from 'chart.js';

ChartJS.register(
  Filler,
  annotationPlugin,
  Title,
  Tooltip,
  Legend,
  LineElement,
  LinearScale,
  PointElement,
  TimeScale,
  TimeSeriesScale,
  LineController,
  Decimation
);
```

Figure 28 Library Imports

As can be seen in Figure 27, Node Package Manager (NPM) was used to successfully install each of the mentioned libraries. Following the completion of the installation, all the necessary components that came along with the libraries were imported into the script tag.

## Application Deployment

The frontend web application was also deployed using another Firebase sub-service which is Firebase Hosting. To start with, the researchers installed firebase-tools via the Node Package Manager (npm).

```
npm install firebase-tools
press h to show help
2:08:44 PM [vite-plugin-svelte] ssr compile in progress ...
2:08:44 PM [vite-plugin-svelte] /src/routes/+layout.svelte:44:1 Unused CSS selector "footer a"
Firebase initialized
2:08:48 PM [vite-plugin-svelte] ssr compile done.
package      files  time  avg
es-lab-air-quality  4  0.36s  90.6ms
2:08:48 PM [vite-plugin-svelte] /src/routes/+layout.svelte:44:1 Unused CSS selector "footer a"
2:08:50 PM [vite-plugin-svelte] ssr compile done.
package      files  time  avg
es-lab-air-quality  2  48.8ms  24.4ms
2:08:50 PM [vite-plugin-svelte] ssr compile done.
package      files  time  avg
es-lab-air-quality  9  0.11s  11.7ms
PS>
D:\...\Lab\es-lab-air-quality npm install -g firebase-tools
[ ] / reify:fsevents: sill reify mark deleted [
```

Figure 29 Firebase tools installation

After the installation of `firebase-tools`, the Svelte web application can be compiled and built via the `npm run build` command, which will generate static HTML and JS files that can readily be deployed to any hosting service.

```
PowerShell
.svelte-kit/output/client/_app/immutable/chunks/singletons-ab1f3f66.js
2.58 kB | gzip: 1.34 kB
.svelte-kit/output/client/_app/immutable/components/pages/_page.svelte-34e8faed.js
3.41 kB | gzip: 1.33 kB
.svelte-kit/output/client/_app/immutable/components/pages/_layout.svelte-37e29201.js
4.67 kB | gzip: 1.87 kB
.svelte-kit/output/client/_app/immutable/components/pages/about/_page.svelte-64b7f50d.js
8.06 kB | gzip: 2.68 kB
.svelte-kit/output/client/_app/immutable/chunks/index-9a4acc40.js
8.31 kB | gzip: 3.35 kB
.svelte-kit/output/client/_app/immutable/start-d1a76aec.js
27.17 kB | gzip: 10.55 kB
.svelte-kit/output/client/_app/immutable/chunks/initFirebase-897470f8.js
47.87 kB | gzip: 10.65 kB
.svelte-kit/output/client/_app/immutable/components/pages/air/_page.svelte-5bc2f69e.js
427.80 kB | gzip: 126.58 kB

> Using @sveltejs/adapter-static
Wrote site to "build"
✓ done
```

Figure 30 Vite Build Step

To prepare the codebase for deployment, firebase needs to initialize its configuration files in the directory of the codebase. The `firebase init` command was used in order to perform this step.

```

PowerShell
Run `npm audit` for details.
1m 12.354s D:\..\..\..\Lab\es-lab-air-quality firebase login
Already logged in as cruz.james99@gmail.com
8.961s D:\..\..\..\Lab\es-lab-air-quality firebase init

##### ## ##### ##### ## ##### #####
## ## ## ## ## ## ## ## ## ##
##### ## ##### ##### ##### ##### #####
## ## ## ## ## ## ## ## ## ##
## ##### ## ## ##### ##### ## ## ##### #####

You're about to initialize a Firebase project in this directory:

  D:\OneDrive - Pamantasan ng Lungsod ng Maynila\4th Year- 1st Semester\Embedded
  Systems\Lab\es-lab-air-quality
? Are you ready to proceed? (Y/n)

```

Figure 31 Firebase Init

The researchers simply followed the prompts for deploying the web application to the correct Firebase project and configuration.

```

PowerShell
? Which Firebase features do you want to set up for this directory? Press Space
to select features, then Enter to confirm your choices. Hosting: Configure files
for Firebase Hosting and (optionally) set up GitHub Action deploys

=== Project Setup

First, let's associate this project directory with a Firebase project.
You can create multiple project aliases by running firebase use --add,
but for now we'll just set up a default project.

? Please select an option: Use an existing project
? Select a default Firebase project for this directory: (Use arrow keys)
> airquality-6d70f (AirQuality)
  back-to-basics-f113d (Back-to-Basics)
  back-to-basics-james (Back-to-Basics-James)
  canis-a4171 (canis)
  gdsc-plm (GDSC-PLM)
  intcognito-advertising (INTcognito Advertising)
  james-lab-78245 (james-lab)
(Move up and down to reveal more choices)

```

Figure 32 Firebase Init

For every time that the researchers wishes to update the live website on <https://airquality-6d70f.web.app/> the `npm run build` and `firebase deploy` commands need to be executed as shown in the image below, showing an upload of 43 files, consisting of the pre-compiled static HTML files.

```
PowerShell
D:\...\Lab\es-lab-air-quality firebase deploy
=== Deploying to 'airquality-6d70f'...
i deploying hosting
i hosting[airquality-6d70f]: beginning deploy...
i hosting[airquality-6d70f]: found 43 files in build
+ hosting[airquality-6d70f]: file upload complete
i hosting[airquality-6d70f]: finalizing version...
+ hosting[airquality-6d70f]: version finalized
i hosting[airquality-6d70f]: releasing new version...
+ hosting[airquality-6d70f]: release complete
+ Deploy complete!
Project Console: https://console.firebase.google.com/project/airquality-6d70f/overview
Hosting URL: https://airquality-6d70f.web.app
```

Figure 33 Firebase Deploy

### III. Results



Figure 34 Live Graphical Representation of Air Quality

The figure shown above is the live graphical representation of air quality that was collected by the sensor and visualized using Svelte and Chart.js on a web-based platform and stored in a Firebase database for future reference. The x-axis represents the value in ppm, y-axis represents the time, and the blue line represents the air quality of a specific area in real time.

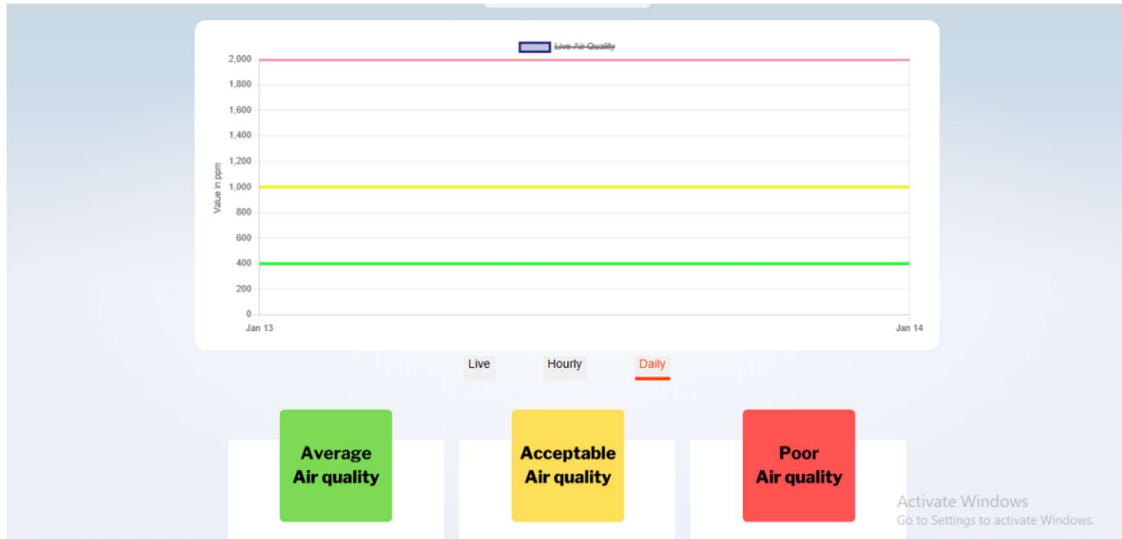


Figure 35 Graphical Representation of Level of CO2 in the Air

The three horizontal lines represent the level of CO2 in the air. As stated by the Wisconsin Department of Health Services [5], the average air quality has a value of 400 in ppm, which is the average outdoor air level, and is represented by green horizontal line in the figure; the acceptable air quality has a value of 400 - 1000 in ppm, which is a typical level found in occupied spaces with good air exchange, and is represented by yellow horizontal line; and the poor air quality has a value of 1000 - 2000 in ppm, which is level associated with complaints of drowsiness and poor air, and is represented by red horizontal line.



Figure 36 Graphical Representation of Daily Air Quality

Additionally, the blue lines show how much carbon dioxide is in a specific area where the sensor collected the data. The higher the CO2 level in a specific area, the lower the amount of fresh air exchange. In this figure, it shows the level of CO2 in the specific area is at the level of acceptable air quality, which has a value of 529.154 ppm in the beginning of the 12th day of January and increases to 990.992 ppm at the end of the 12th day of January.



Figure 37 Graphical Representation of Hourly Air Quality

Lastly, it shows in this figure the level of CO2 hourly in the specific area, which consistently at the level of acceptable air quality until 1pm wherein the level of CO2 increases to 1035 ppm and decreases to 947.592 ppm at 2pm.

## IV. Conclusion

In conclusion, this research project has successfully achieved its goal of creating an IoT-based air quality monitoring system that is not only cost-effective but also user-friendly. By utilizing the ESP32 microcontroller and the MQ135 air quality sensor, the system can detect a wide range of gases in the air and provide real-time data visualization to users via web.

The system represents a significant step forward in the fight against air pollution, providing individuals and communities with the necessary information to make informed decisions about their environment. With the increasing need for air quality monitoring, this system offers a solution that is both practical and accessible.

Furthermore, the project has met not only its technical objectives but also its educational ones by providing the opportunity to apply the principles and techniques acquired in the Embedded Systems Laboratory course.

As we look to the future, the hope is that this system will serve as a model for other communities facing similar challenges and will help to create a cleaner and healthier environment for all. This project serves as a reminder that with determination and innovation, we can work towards a better future for ourselves and future generations.



## References

- [1] J. Fenger, “Air pollution in the last 50 years – From local to global,” *Atmos. Environ.*, vol. 43, no. 1, pp. 13–22, Jan. 2009, doi: 10.1016/j.atmosenv.2008.09.061.
- [2] N. Agarwal *et al.*, “Indoor air quality improvement in COVID-19 pandemic: Review,” *Sustain. Cities Soc.*, vol. 70, p. 102942, Jul. 2021, doi: 10.1016/j.scs.2021.102942.
- [3] Elprocus, “MQ135 Air Quality Sensor : Pin Configuration, Working & Its Applications.” <https://www.elprocus.com/mq135-air-quality-sensor/> (accessed Jan. 13, 2023).
- [4] DroneBot Workshop, “Getting started with the ESP32-CAM,” 2023. <https://dronebotworkshop.com/esp32-cam-intro/> (accessed Jan. 13, 2023).
- [5] Wisconsin Department of Health Services, “Carbon Dioxide,” 2022. [https://www.dhs.wisconsin.gov/chemical/carbondioxide.htm?fbclid=IwAR3B2CH3QkzPXUpBRoRTPywqXjfOQWua0VvvGDr6pCugGF4ZRz\\_dPsCoUoQ#:~:text=400-1%2C000 ppm%3A typical level,stagnant%2C stale%2C stuffy air](https://www.dhs.wisconsin.gov/chemical/carbondioxide.htm?fbclid=IwAR3B2CH3QkzPXUpBRoRTPywqXjfOQWua0VvvGDr6pCugGF4ZRz_dPsCoUoQ#:~:text=400-1%2C000 ppm%3A typical level,stagnant%2C stale%2C stuffy air) (accessed Jan. 13, 2023).
- [6] G. Krockner, “Arduino Library for the MQ135.” Accessed: Dec. 30, 2022. [Online]. Available: <https://github.com/GeorgK/MQ135>
- [7] R. Lindsey, “Climate Change: Atmospheric Carbon Dioxide,” *Climate.gov*, 2022. <https://www.climate.gov/news-features/understanding-climate/climate-change-atmospheric-carbon-dioxide> (accessed Jan. 13, 2023).
- [8] PlatformIO, “What is PlatformIO?” <https://docs.platformio.org/en/latest/what-is-platformio.html> (accessed Jan. 13, 2023).
- [9] Last Minute Engineers, “ESP32 Basic Over The Air (OTA) Programming In Arduino IDE,” 2022, Accessed: Jan. 13, 2023. [Online]. Available: [https://lastminuteengineers.com/esp32-ota-updates-arduino-ide/#:~:text= \(OTA\).-,What is OTA programming in ESP32%3F,access to the ESP module.](https://lastminuteengineers.com/esp32-ota-updates-arduino-ide/#:~:text= (OTA).-,What is OTA programming in ESP32%3F,access to the ESP module.)
- [10] A. Sharma, “Elegant OTA.” 2019. Accessed: Jan. 13, 2021. [Online]. Available: <https://www.arduino-libraries.info/libraries/elegant-ota>
- [11] Random Nerd Tutorials, “ESP32 OTA (Over-the-Air) Updates – AsyncElegantOTA using Arduino IDE,” 2023. <https://randomnerdtutorials.com/esp32-ota-over-the-air-arduino/> (accessed Jan. 13, 2023).
- [12] S. Philip, “What is Svelte? & Why it is so popular?,” 2023.

<https://www.zyxware.com/article/what-is-svelte-why-is-it-so-popular> (accessed Jan. 13, 2023).

[13] Educative Answers Team, “What is Svelte?,” *Educative*.

<https://www.educative.io/answers/what-is-svelte> (accessed Jan. 13, 2023).

[14] Chart.js, “Chart.js,” 2023. <https://www.chartjs.org/docs/latest/> (accessed Jan. 13, 2023).

[15] GeeksforGeeks, “Moment.js,” 2022. <https://www.geeksforgeeks.org/moment-js/> (accessed Jan. 13, 2023).

# Appendices

## ESP32-CAM Firmware Source Code

```
main.cpp
#include <Arduino.h>
#include <time.h>

#include <WiFi.h>
#include <WebSerialLite.h>
#include <ESPAsyncWebServer.h>
#include <AsyncTCP.h>
#include <AsyncElegantOTA.h>

#include <MQ135.h>

#include <Firebase.h>
#include <FirebaseJson.h>
#include <addons/TokenHelper.h>
#include <addons/RTDBHelper.h>

#define LED_BUILTIN 33
#define GAS_SENSOR 15
#define OTA_MODE_EN 13

const char *host = "esp32";
const char *ssid = "HUAWEI-V4XU";
const char *password = "Tataycruz";

const char *firebaseApiKey = "AIZaSyC9ptugG-7U8EcTXQk-5nCje30vAvrXHMw";
const char *firebaseDbUrl = "https://airquality-6d70f-default-rtdb.asia-southeast1.firebaseio.com/";

FirebaseData fb_dataObject;
FirebaseAuth fb_auth;
FirebaseConfig fb_config;
bool signupOk = false;

AsyncWebServer server2(80); // HTTP server descriptor assigned to port 80,
used by the OTA and webserial server

int otaFlashMode;

struct sensorValue_t {
    int rawAnalogRead;
    float ppmReading;
    tm readAt;
};

MQ135 sensor(GAS_SENSOR);
float r0Calibration;
// int sensorValues[3];
sensorValue_t sensorValues[3];

String lastFirebaseMsg = "0";
String lastFirebaseErr = "0";

void readGasSensor() {
    digitalWrite(LED_BUILTIN, 1);
    // The ADC of the GPIO pins are shared with the ADC used by the
    // WiFi antenna. To use analogRead, we first need to disconnect
    // WiFi and then reconnect after
    WiFi.disconnect(true, false);

    for (int i = 0; i < 3; ++i) {
        sensorValues[i].rawAnalogRead = analogRead(GAS_SENSOR);
        sensorValues[i].ppmReading = sensor.getPPM();
        getLocalTime(&sensorValues[i].readAt);
    }
}
```

```

    delay(1000);
}

WiFi.begin(ssid, password);
// wait for connection
while (WiFi.status() != WL_CONNECTED) {
    digitalWrite(LED_BUILTIN, 0);
    delay(250);
    digitalWrite(LED_BUILTIN, 1);
    delay(250);
}
digitalWrite(LED_BUILTIN, 0);
}

void receiveWebSerial(uint8_t* data, size_t len) {
    String d = "";
    for(int i = 0; i < len; i++){
        d += char(data[i]);
    }

    webSerial.println(d);

    if (d.equals("reboot")) {
        webSerial.print("Device restarting ...");
        webSerial.print("Refresh this page if it does not automatically
reconnect");

        digitalWrite(LED_BUILTIN, HIGH);
        delay(1000);
        esp_restart();
    }

    if (d.equals("mode")) {
        // Implement Serial command that shows current OTA mode for debugging
        purposes
        if (otaFlashMode == 1) {
            webSerial.print("MODE: OTA flash mode");
        } else {
            webSerial.print("MODE: Run mode");
        }
        webSerial.println();
    }

    if (d.equals("sensor read")) {
        // Implement serial command that forces sensor read. This will
        // momentarily turn off WiFi
        readGasSensor();
    }

    if (d.equals("sensor show")) {
        // Implement serial command that shows contents of sensorValues array

        for (int i = 0; i < 3; ++i) {
            webSerial.print("RAW: ");
            webSerial.println(sensorValues[i].rawAnalogRead);
            webSerial.print("PPM: ");
            webSerial.println(sensorValues[i].ppmReading);
            webSerial.print("TIME: ");
            tm tempTime = sensorValues[i].readAt;
            webSerial.println(asctime(&tempTime));
        }
    }

    if (d.equals("sensor r0")) {
        webSerial.print("R0: ");
        webSerial.println(r0Calibration);
    }

    if (d.equals("time")) {
        struct tm tempTime;
        getLocalTime(&tempTime);
    }
}

```

```

    webSerial.println(asctime(&tempTime));
}

if (d.equals("firebase status")) {
    webSerial.print("Ready: ");
    webSerial.println(Firebase.ready());
    webSerial.print("SignupOK: ");
    webSerial.println(signupOk);
}

if (d.equals("firebase last")) {
    webSerial.print("Msg: ");
    webSerial.println(lastFirebaseMsg);
    webSerial.print("Err: ");
    webSerial.println(lastFirebaseErr);
}
}

void initializeWifiAndOTA() {
    // Connect to WiFi network
    WiFi.mode(WIFI_STA);
    WiFi.begin(ssid, password);
    Serial.println("");

    // Wait for connection
    while (WiFi.status() != WL_CONNECTED) {
        delay(500);
        Serial.print(".");
    }

    Serial.println("");
    Serial.print("Connected to ");
    Serial.println(ssid);
    Serial.print("IP address: ");
    Serial.println(WiFi.localIP());

    // Basic index page for HTTP server, HTTP server will be
    // used for OTA and WebSerial
    server2.on("/", HTTP_GET, [](AsyncWebServerRequest *request) {
        request->send(200, "text/plain", "/update for OTA upload, /webserial
for serial monitor");
    });

    // Elegant Async OTA
    // Start OTA server so we don't need USB cable to upload code
    AsyncElegantOTA.begin(&server2);
    server2.begin();

    // Start webserial server so we don't need USB cable to view serial
monitor
    webSerial.onMessage(receiveWebSerial);
    webSerial.begin(&server2);
}

void setup(void) {
    pinMode(OTA_MODE_EN, INPUT);
    pinMode(GAS_SENSOR, INPUT);
    pinMode(LED_BUILTIN, OUTPUT);

    if (digitalRead(OTA_MODE_EN) == 0) {
        // Pull GPIO 13 low if you want to enable to ota flash mode
        otaFlashMode = 1;
    } else {
        otaFlashMode = 0;
    }

    // Calibrate and test the sensor before WiFi is turned on
    // because we can no longer use analogRead once WiFi radio is on
    r0Calibration = sensor.getRZero();
    // for (int i = 0; i < 3; ++i) {
    //     sensorValues[i].rawAnalogRead = analogRead(GAS_SENSOR);
}

```

```

// sensorValues[i].ppmReading = sensor.getPPM();
// delay(1000);
// }

Serial.begin(115200);
Serial.println("Serial connected");

// Since our ESP32-CAM does not have a micro USB connector, there are 2
options:
// 1.) Use a separate programmer board that plugs in to the TX and RX
pins
// of the ESP32, to upload code and view the serial monitor
// or,
// 2.) utilize the wifi capabilities of the ESP32 to upload code over-
the-air (OTA)
// We have chosen this option for convenience and cost savings
initializeWifiAndOTA();

if (otaFlashMode == 0) {
// OTA flash mode is a failsafe. If it is on, only OTA will work,
everything
// will be disabled. This ensures that we can reupload code to the
device even
// if we upload broken firmware to it. If OTA flash mode is off, OTA
and everything else
// will still work, but OTA might not be fast or reliable
const long GMT_OFFSET_PH = 8;
const char* NTP_SERVER1 = "pool.ntp.org";
const char* NTP_SERVER2 = "time-a-g.nist.gov";
const char* NTP_SERVER3 = "time.google.com";
configTime(GMT_OFFSET_PH, 0, NTP_SERVER1, NTP_SERVER2, NTP_SERVER3);

fb_config.api_key = firebaseApiKey;
fb_config.database_url = firebaseDbUrl;
fb_config.token_status_callback = tokenStatusCallback;
fb_auth.user.email = ""; // Anonymous login
fb_auth.user.password = "";

// Login to fb
if (Firebase.signUp(&fb_config, &fb_auth, fb_auth.user.email,
fb_auth.user.password)) {
signupOk = true;
}

Firebase.begin(&fb_config, &fb_auth); // Connect to db
// Don't let firebase library control wifi auto connect,
// we want to control this on our own
Firebase.reconnectWiFi(false);
Firebase.setDoubleDigits(5);
}

// If all goes well, turn on the built-in red LED of the ESP32
// Note that LED_BUILTIN has reverse logic, writing LOW to it turns it on
digitalWrite(LED_BUILTIN, LOW);
}

void writeToFirebase() {
// Since sensor reading is done three times at a time, we can get the
average
// of the last sensor readings to mitigate errors in the data
float averagePpmSample = (sensorValues[0].ppmReading +
sensorValues[1].ppmReading +
sensorValues[2].ppmReading) / 3;

int averageAnalogValuesSample = (sensorValues[0].rawAnalogRead +
sensorValues[1].rawAnalogRead +
sensorValues[2].rawAnalogRead) / 3;

// const char* path = "devices/james-esp32/";
tm tempTime = sensorValues[0].readAt;

```

```

FirebaseJson obj;
obj.add("ppm", averagePpmSample);
obj.add("rawAnalog", averageAnalogValueSample);
obj.add("readAt", mktime(&tempTime));

if (Firebase.RTDB.pushJSON(&fb_dataObject, "devices/james-esp32", &obj))
{
  lastFirebaseMsg = fb_dataObject.dataPath();
} else {
  lastFirebaseErr = fb_dataObject.errorReason();
}
}

unsigned long millisSinceLastReady = 0;

void loop(void) {
  if (otaFlashMode == 0 && (millis() - millisSinceLastReady > 30000 ||
  millisSinceLastReady == 0)) {
    webSerial.println("Reading gas sensor, temporarily turning off WiFi");
    readGasSensor();
    if (Firebase.ready() && signupok) {
      // Firebase.ready() should be repeatedly called but with a set
      interval // of 15s to prevent spam. We can't use the delay() function because
      that // blocks the CPU and disables other function calls, hence we use the
      millis // timer, and manually check for every 30th second, we send
      Firebase.ready() and other
      // fb related operations
      writeToFirebase();

      millisSinceLastReady = millis();
    }
  }
}
}

```

### Svelte Web Application Source Code

For a complete overview of the frontend repository, visit <https://github.com/jamesc2000/es-lab-air-quality> .