Vietnamese – German University
Electrical and Computer Engineering Study Program

Frankfurt University of Applied Science
Faculty 2: Computer Science and Engineering

Design and Implementation of a Modular IoT-Based Environmental Monitoring System Using ESP32

Group █

HỒ HỮU KHÁNH AN ████████████
ĐẶNG QUỲNH HƯƠNG ████████████
LÊ NHÂN ÁNH DƯƠNG ████████████

MODULE PROJECT

Submitted in partial fulfillment of the requirements for module Object-oriented Programming (OOP) in study program
Electrical and Computer Engineering,

Vietnamese – German University, █████

ABSTRACT

Environmental monitoring is crucial for understanding the local air quality and climate; however, most available approaches are quite expensive, lack the versatility to be used in academia and experimentation. This is the low-cost platform project, suitable for students and hobbyist alike who want to make a modular environmental monitoring system based on the ESP32 microcontroller. The unit is equipped with a PMS7003 particulate matter sensor, a BME/BMP280 sensor for monitoring temperature, humidity and barometric pressure, and a NEO-M8N module GPS tracker for real-time reference.

The firmware is in C/C++ language, which allows an object-oriented programming (OOP) approach that guarantees a reusable and well-organized system. The sensor readings are collected periodically, validated, and transmitted to a web server for storage and visualization. The data is displayed in a local server as well as a cloud-based service, which is publicly available, with a proposal for an accompanied real-time GPS tracking capability on a test unit when it deploys in different locations. We anticipate that a functional and scalable monitoring node will result for a system architecture, which in the future can be grown to support multiple deployable units and additional higher quality sensors.

MEMBER TASKS

GROUP ▮ – OOP Module – ▮▮▮	
Student Name	**Responsibilities**
Hồ Hữu Khánh An	**Programmer**: main file and sensor's implementations. Purchase components and assembly. Proof checking programs. Uploading and conducting testing phases.
Đặng Quỳnh Hương	**Programmer**: header files and sensor's implementation check. Uploading program. Conducting system auxiliary tests.
Lê Nhân Ánh Dương	**Writer and Support**: Drafting report contents. Proof checking grammar and diagrams. Conducting system auxiliary tests.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

CHAPTER 1: INTRODUCTION

Environmental monitoring has become more applicable in recent years due to increasing concerns about air pollution and its effects on human health. In residential environments, airborne particulate matter, especially finer particles such as PM2.5 and PM10, has proven to pose significant health risks at higher concentration. However, commercial systems are often expensive, lack flexibility, or unsuited for educational purposes. This has led to a growing demand for low-cost solutions that is amendable for indoor and semi-indoor uses, while remaining accessible to students and hobbyists.

This project focuses on the design and implementation of an IoT-based air quality monitoring system using an ESP32 platform. The system is primarily developed following object-oriented methods while maintaining a formal and simple engineering approach. The core functionality is the real-time measurement of particulate matter concentrations using PMS7003 sensor. Additional parameters, including geographic location, are integrated to provide contextual information for the displayed data in URL servers. Temperature, humidity and atmospheric sensing via a BME280 sensor are considered as a secondary part of the project.

This monitoring system is intended both indoor and outdoor, such as residential spaces, laboratories, factories, and areas with Wi-Fi connectivity. It can also be housed in 3D-printed or commercial weather-proof containers. Wireless communication enables sampling without physical retrieval, while modular design is explored to allow selected components to be adapted with minimal issue. The scope of this work is limited to the development of a functional prototype rather than large-scale field analysis.

This report is divided into structured chapters. Chapter 1 provides an introduction to the content and its objectives. Chapter 2 presents a review of related literatures from existing systems related to this IoT application. Chapter 3 describes and analyzes the electrical system. Chapter 4 examines the performance, including analysis to the data and website functionality. Chapter 5 discusses results, limitations, and potential improvements to hardware and software, while chapter 6 closes up the report. Finally, appendices are provided containing necessary object-oriented programming, which is also displayed in a linked GitHub repository.

CHAPTER 2: LITERATURE REVIEW

2.1 Overview of IoT Air Monitoring Systems

A large number of recent students highlights a growing interest in low-cost Internet of Things (IoT) solutions for air quality monitoring. This is partly arises in response to the need for accessible, scalable, and real-time environmental data collection. However, conventional air monitoring stations with high accuracy are often expensive and lack deployment flexibility (Othman et al., 2024; Taştan, 2022). Consequently, the interest in designs and implementation of compact sensor-based alternatives for tracking pollutants such as particulate matter PM2.5/PM10 in localized environments has increased. These systems aim to balance affordability and performance while supporting continuous and reliable data acquisition.

Particulate matter is one of the most commonly monitored air quality parameters due to its direct impact on human health. Cheap optical sensors, such as the PMS-series, are widely used in research-oriented monitoring systems because of their digital output, compact form factor, and reasonable performance for indicative measurements. Although they do not achieve reference-grade accuracy, studies show that they are suitable for trend analysis and comparative monitoring when properly calibrated (Kim et al., 2023; Siddiqui et al., 2024). Several studies also emphasize the importance of data accessibility and user awareness in air quality monitoring applications. IoT-based systems often incorporate wireless communication and web-based visualization to allow users to observe environmental conditions (Othman et al., 2024). These approaches establish a common foundation for low-cost systems and inform the overall direction of this project.

Figure 1: *Examples of household and industrial-grade air quality stations;*

2.2 ESP32 Data Acquisition and Integration

The ESP32 microcontrollers and its derivatives rank among the most popular choices in IoT system designs, thanks to its Wi-Fi capability, low cost, and adequate processing power. Comparing to similar platforms like Arduinos, the ESP32 allows easy internet connectivity without external modules, thereby reducing complexity and power requirement (Pineda-Tobón et al., 2024; Omkar et al., 2024). This makes it particularly suitable for compact air quality monitoring stations. In existing research, ESP32 can be easily interfaced with PMS devices, environmental sensors, and positioning modules via standard communication protocols. PMS devices and GPS usually communicate via UART, while temperature, humidity, and pressure sensors often rely on I^2C interface (Pineda-Tobón et al., 2024). Such standardized protocols allows additional sensors to be added without significant changes to the core architecture.

Location and time information are frequently integrated into IoT monitoring systems to enhance data interpretation. GPS positioning enables geographic tagging of measurements, while network-based time synchronization provides accurate timestamps without additional hardware (Omkar et al., 2024). Previous studies indicate that combining environmental data with spatial and temporal context improves the usefulness of monitoring systems, particularly for comparative analysis across different deployment locations. These findings directly influence the system design adopted in this project's integration using ESP32-based platforms.

2.3 Summary of Relevant Literature

From previous references, we conclude that low-cost IoT air quality sampling stations with remote data monitoring are highly suitable for local and indoor applications, particularly for measuring particulate matters via compact optical devices. While these systems do not provide a high accuracy result, prior researches confirms their effectiveness for monitoring and trend analysis applications in healthcare and quality-of-life improvement. The ESP32 is widely adopted in such systems due to its integrated wireless connectivity and support for modular communication interfaces. Additionally, the inclusion of contextual information such as location and time has been shown to improve data interpretation and comparison across deployments. Based on these findings, this project applies an ESP32-based modular design to implement a practical environmental monitoring solution that balances cost, functionality, and educational value.

CHAPTER 3: ELECTRICAL SYSTEM DESIGN

3.1   System Architecture Overview

Our proposed system is a low-cost IoT environmental monitoring platform designed for indoor particulate data acquisition and wireless transmission. We use an ESP32 DevKit V1 microcontroller operating on 2.4 GHz Wi-Fi to serve as the communication and processing unit. The ESP32 is mounted on a 30-pin three-row expansion shield, enabling modular connections and adequate power supply via jumper wires while maintaining flexibility for system expansion. The sensing subsystem includes a PMS7003 optical particulate matter sensor for PM2.5 and PM10 measurement, a BME280 for ambient temperature, humidity, and barometric pressure monitoring, and a NEO-M8N with GPS antenna for location acquisition. The PMS7003 and GPS communicate with the microcontroller through UART, while the BME280 operates via the I²C protocol. These data are periodically sampled and processed for wireless transmission to a server. The entire system is housed inside a 3D-printed container with its dimension provided in *Table 1*.

PROJECT: ESP32 AIR QUALITY MONITORING STATION		
FEATURE	**UNIT OF MEASUREMENT**	**VALUE**
Dimensions: L x W x H (when put inside a container)	Millimeter (mm)	95 x 85 x 40
Total Mass (no container and USB wire)	Kilogram (Kg)	0,91
Micro-USB Wire Length	Meter (m)	1
Voltage Supply	Volt (V)	5
Current Supply	Ampere (A)	2
Current Consumed	Ampere (A)	1,2
Number of Integrated Components	Module	3
Wi-Fi Band Requirement	Giga Hertz (GHz)	2.4
Wi-Fi Connection Rage (for ESP32)	Meter (m)	~85
Indoor Operational Time	Assumed plugged in all time	Unrestricted
Outdoor Operational Time (Recommended, outdoor increases the chance of catching GPS signal)	Hour (h), with cooling	Unrestricted
	Hour (h), without cooling	~48

Table 1: *Basic system statistics and power requirements;*

The above table provides the key physical and operational specifications for the proposed environmental station. We design the system to support both local and remote data visualization. A local web server hosted on the ESP32 allows nearby devices to monitor real-time through a HTML browser accessible within the same network. New data is updated on this browser with an interval of approximately 4 seconds and restrict same server devices to access the URL. In addition, the collected data are uploaded to a ThingSpeak cloud platform with an interval of 15 seconds, enabling remote access and historical visualization without connecting to the local internet. Our system is powered by a 5V Micro-USB supply rated up to 2A, making this suitable for continuous indoor operation as well as limited outdoor deployment with a battery.

3.2   Components and Operational Diagrams

The components for this environmental IoT system is chosen based on considerations for cost efficiency, ease of integration, and suitability for a variety of applications. We selected the ESP32 DevKit V1, also known as NodeMCU LuaNode 32, to be the project's main processing unit due to its integrated 2.4 GHz Wi-Fi and native support for common communication protocols such as UART and I^2C. The program implementations are uploaded directly to the microcontroller USB port. These features allow our team to reliably interface with each sensor, design the URL display, integrate ThingSpeak illustrations, as well as transmit data without requiring additional hardware. The below *Table 2* lists the electrical components, their general statistics, and purchase cost per unit. Their capabilities and prices are our motivation to choose them for the project.

COMPONENT	QUANTITY	GENERAL SPECIFICATION	COST / UNIT
BLE ESP32 NodeMCU LuaNode32 RF Module 30Pin (DevKit V1)	1	– Dual-core 32-bit Xtensa LX6 – Operating voltage: 3.3 V (5 V input via Micro-USB) – Integrated 2.4 GHz Wi-Fi (802.11 b/g/n) and Bluetooth – 240 MHz CPU frequency	115,000 đ
Breakout Shield ESP32 DEVKIT V1 30 Pin	1	– Breakout board compatible with ESP32 DevKit V1 – Exposes all 30 GPIO male pins – Includes power, ground, and three I/O rows with labeling – Supports stackable expansion for modules and shields	32,000 đ

PMS7003 Laser Optical Particle Detection with G7 Adapter	1	– Laser scattering for detection – Measures PM1.0, PM2.5, and PM10 mass concentrations – Digital UART serial output – Operating voltage: 5 V	**450,000 đ**
GY-BME280 SPI/I²C Humidity, Temperature, Pressure Sensor	1	– Triple sensor: Temperature, Humidity, Barometric Pressure – Communication: I²C and SPI – Operating voltage: 1.8–3.6 V – Compact breakout module	**145,000 đ**
Module GPS NEO-M8N with active/passive GPS antenna	1	– Multi-GNSS receiver: GPS, GLONASS, Galileo, QZSS – UART interface – Operating voltage: 3.3–5 V	**157,000 đ**
5V 2A Charger Adapter (<u>Recommended:</u> *Battery for outdoor deployment*)	1	– For static indoor use, any reliable 5V charger adapter – For portable outdoor use, switch to 5V 10Ah batteries	**30,000 đ**

Note – *The cost-per-unit column depends on the each supplier's price. The table does not consider extra module replacement cost nor the shipping fees. The table excludes the cables and breadboard prices.*

Table 2: *Electrical components table and price tag per unit;*

For particulate matter monitoring, we selected the PMS7003 sensor due to its digital measurement of PM2.5 and PM10 concentrations and its proven low-cost reliability. Ambient temperature, humidity, and barometric pressure are measured using BME280, which provides multiple parameters within a very compact module. Location and time logs are obtained from the NEO-M8N GPS board to add spatial context to the collected data. Since the PMS7003 and the GPS module required a steady 5V power supply, an ESP suitable 30-pin breakout shield is chosen to simplify wiring and enhance system modularity to all ESP32 input and output pins.

The logical operation of the system is illustrated in the below ***Figure 2***, showing the interaction flow between the sensing devices, the ESP32 microcontroller, and the data visualization platforms. The ESP32 is responsible for preprocessing the sensor data and transmit either to a 2.4 GHz suitable local web interface or to a cloud-based platform for remote monitoring, which in this project is the free ThingSpeak hosting server.
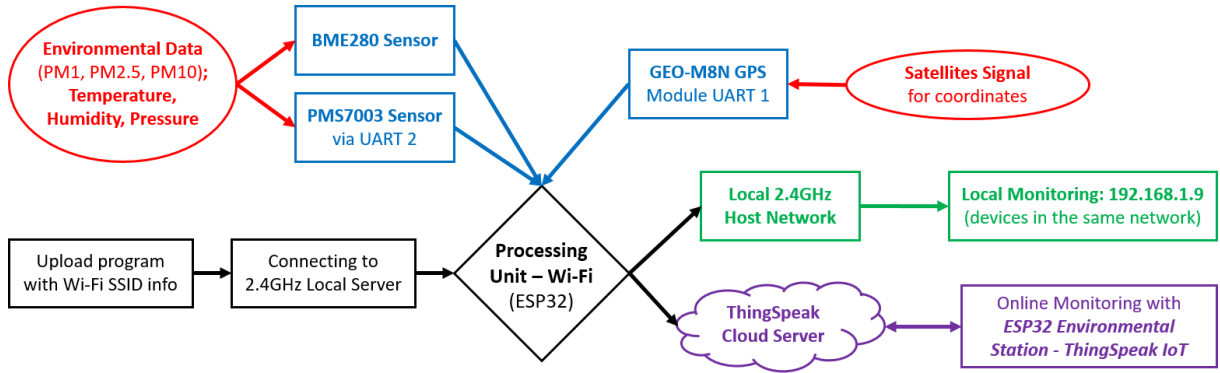
Figure 2: *System architecture and operational flow of the project;*
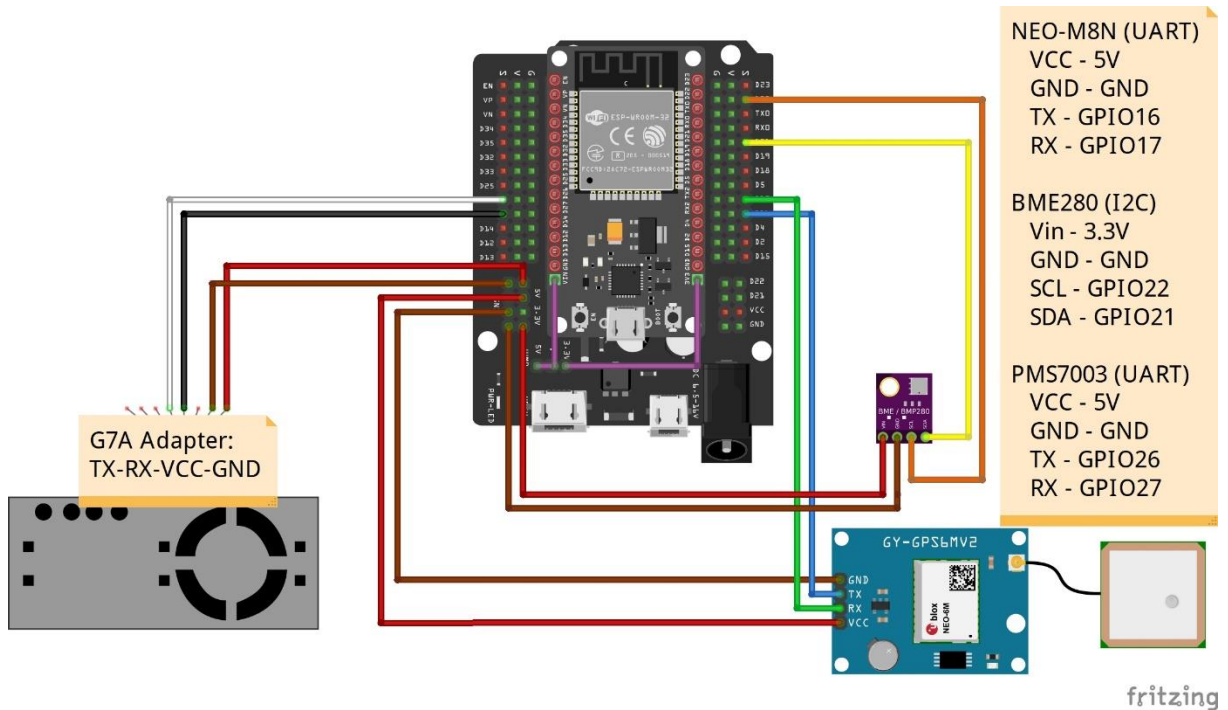
Figure 3: *The wiring diagram for the air quality monitoring system;*

The physical wiring and signal connections are made via Fritzing software in ***Figure 3***, which serves as a reference for the assembly by outlining power distribution and communication interfaces. Together, these design elements and operational design support a modular, reliable foundation for an environmental monitoring system suitable for educational and practical deployment. The next chapter analyzes the performance, power consumption and recommendation as well as discusses the project implementation in different conditions.

CHAPTER 4: IMPLEMENTATION AND ANALYSIS

4.1 System Assembly and Programming

We assemble the hardware by firstly mounting the ESP32 DevKit V1 on the 30-pin breakout expansion shield, the shield sole purpose is to facilitate 5V power distribution and organizes sensor connections while the ESP with its micro USB will be the only uploading and debugging interface. The GPS module (NEO-M8N) is communicated via UART1 using the processor's GPIO16 for the GPS's transmitted (TX) and GPIO17 for the receiver (RX), while the PMS7003 particulate sensor interfaces via UART2 on the processor's GPIO26 for its TX and GPIO27 for RX. The BME280 sensor communicates using I²C, with SDA on GPIO21 and SCL on GPIO22, and powered with 3.3 V. Care was taken to align sensor power and signal pins correctly on the shield's labeled Ground (G), 3.3 V (V), and Signal (S) rows to avoid damages. The assembly is inserted into a specially designed 3D-printing container as shown in *Figure 4*. The container is colored orange, the hatches for top and bottom is in red and use eight M3 screws. Design figures and its file are in this TinkerCAD link: ***Container design with 2 hatches***.
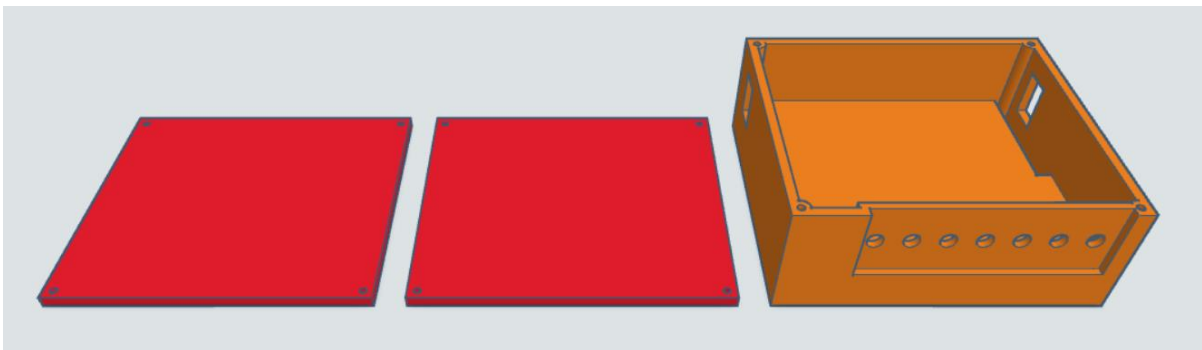
Figure 4: *The design for the 3D-printed container;*

The implemented firmware is written in C/C++ language on ***Visual Studio Code*** and are uploaded thanks to PlatformIO extension. We designed the programs to follow an object-oriented approach by encapsulating sensor handling and server routine to separate classes, which improves debugging and maintainability. We also integrated necessary libraries to support sensors, including a ***PMS7003 library*** obtained from GitHub, an ***Adafruit BME280 driver*** for environmental measurements, and ***TinyGPSPlus*** for parsing GPS signal. The ESP32 first task is connecting to the configured local Wi-Fi network. Then it initializes each sensor interface and starts both the local web server and cloud upload tasks.

4.2 Power Consumption Figures

In order to estimate the electrical requirement, the maximum current consumption is considered to determine the corresponding power demand. Knowing the estimated power and the supply voltage U, we can find an ideal current I figure. This relationship is expressed as:

$$P_e = U \times I_M$$

Where P_e (W) is the electrical power consumption, I_M (A) is the consumed current, and U (V) is the stable applied voltage. Since this system operates from a regulated 5V supply, variations in power usage are primarily influenced by changes in sensor activity and wireless communication. For portable outdoor deployment using a standard 5V power bank, the expected operating duration can be approximated based on the battery capacity. Given a battery rated at capacity C (*mAh*), the theoretical runtime T (*hours*) under average load can be estimated as:

$$T \approx C / I_M$$

This estimation assumes a stable voltage and does not account for conversion losses. The system's total current used and thermal lost during normal operation is found by:

$$I_{total} = I_{consumed} \approx I_{ESP32} + I_{BME} + I_{PMS} + I_{GPS} + I_{thermal}$$
$$\approx 240 + 0{,}003 + 100 + 50 + 2 \approx 400 \text{ (mA)}$$

The information given in *Table 3* provides a practical deployment duration reference for selecting appropriate power banks such as those linked accordingly. Adequate airflow around the PMS7003 inlet and proper cooling are recommended during prolonged outdoor operation.

Battery Statistics with Examples (estimated constant flow, normal load, with cooling)	Maximum Operational Duration (approximate)
5V 5000mAh battery (Example: *Duracell Charge 5V 5000mAh Slimline Power Bank DMP-PB-CHARGE | B&H*)	12 hours 30+ minutes
5V 10000mAh battery (Example: *PB02-02 PD22.5W 10000mAh 1C1A Digital Display Power Bank - Trusmi*)	25+ hours
5V 15000mAh battery (Example: *5V Extended Life 15000mAh Power Bank – ActionHeat Heated Apparel*)	37+ hours
5V 20000mAh battery (Example: *Polymer Power Bank 20000mAh Type C 10.5W AVA+ G-DX258 | thegioididong*)	50+ hours

Table 3: *Operational power consumption figure for outdoor deployment;*

4.3   Code Implementation Logic

Fundamentally, our firmware is structured using C/C++ classes with each corresponding to a physical subsystem of the aforementioned station. These classes act as wrappers around existing hardware libraries to provide a consistent interface for data recollection. For the full codes, you can access our **project GitHub**, which includes more explanations with `.github/workflows` and `.gitignore` files for workflow configuration in PlatformIO. Some of the programs are also listed in the Appendix sections at the end of this report.

We also created a `BME280` class for environmental sensing via Adafruit BME280 library. It encapsulates temperature, humidity, and pressure readings and stores them as private variables. The `begin()` method initializes I^2C, while the `update()` reads fresh sensor data. Getter methods (`temperature()`, `humidity()`, and `pressure()`) return cached values efficiently. Meanwhile, a `GPS` class handles location from the NEO-M8N. Its constructor binds the class to a specific `HardwareSerial` for UART assignment. The `update()` in this header file continuously decodes serial data, while `hasFix()`, `latitude()`, `longitude()`, and `satellites()` provide structured access to parsed GPS information.

We also use a `PMS7003` class to manage UART matter sensing. It wraps the PMS library and extracts PM1.0, PM2.5, and PM10 from the received data frames, the `update()` here returns a boolean status to indicate successful data acquisition. A `WebServerUI` class is uniquely developed for this project and manages local data visualization. It aggregates references to all sensor objects and the ESP32 web server through its constructor. The `begin()` method registers HTTP routes, while a private logic serves a dynamic HTML page with latest sensor readings.

The ESP32 first task is connecting to the configured local Wi-Fi network. Then it initializes each sensor interface and starts both the local web server and cloud upload tasks. Efficient data handling is essential to ensure that our measurements remain accessible and meaningful. In the next section, we examine the data acquisition and visualization mechanisms implemented through local and cloud web servers.

4.4   Data Acquisition via Web Servers

The sensor data are delivered through two output pathways. First, a locally hosted web browser on the connected local 2.4 GHz Wi-Fi, and second is a cloud platform. We implement the local interface, i.e. UI, using the ESP32 `<WebServer.h>` library. It is designed to display the real-

time sensor values and GPS coordinates data, satellite count, and a Google Maps hyperlink button for immediate location visualization. Local updates are performed periodically each four seconds via a basic *HTML meta refresh mechanism*. This enables nearly simultaneous monitoring for devices within said local area network.

As a secondary backup, we also allow remote access and long-term observation. This is done by uploading sensor data to the ThingSpeak platform using `<http.GET();>` request at a fixed 15-second interval. On our project's ThingSpeak, which can be accessible via this attachment: ***ESP32 Environmental Station - ThingSpeak IoT***, each environmental parameter is assigned to an individual channel which allows continuous time-series visualization as well as record export. During testing phases, the transmission process remained stable without noticeable packet loss or abnormal latency. We, of course, encountered temporary data interruptions during power cycling and when pressing EN (enable) button on ESP32, after which the system resumed operation following sensor initialization and calibration delays.

4.5   Troubleshooting and Summary

Despite a relatively calm testing phase, a few minor technical issues were identified during implementation and they were resolved through each review. We encountered an initial UART configuration error, which prevented stable GPS acquisition. This was corrected by reassigning the appropriate transmit and receive pins. Additionally, brief calibration delays were observed for the PMS7003 and BME280 after power cycling. Based on manufacturer specifications, we concluded it is a known characteristic these low-cost modules. Temporary data interruptions only occurred when the system was unpowered, especially when it is being moved to new destinations.

An important note to mention about troubleshooting local server operation with the ESP32 is that when prolonged power disconnection occurs, when the power is supplied again, there is a high chance the the local URL address is automatically changed. During sudden power loss testing for continuous one hour, we found out that although the system still works properly on ThingSpeak server, it is not possible for us to reconnected to the local website on any devices. Later debugging pointed out the the local address has changed its last two digits. The base template for the local address is **192.168.xx.xx** and before the test was **192.168.1.9**, afterward, it changed to **192.168.1.2**. We compared such characteristic to known reports from manufacturer and on various IoT forums. This led us to the conclusion that the current local architecture with the ESP32 is not robust enough

when facing long-term power loss, which also push us to find alternatives for making the controller becomes its own login server in addition to relying on the configured routers.

ADVANTAGES	DISADVANTAGES
– A low-cost and modular architecture – Highly stable real-time observation – Enable both cloud and local access – Simple system assembly – Relatively clear data presentation	– Limited to 2.4 GHz networks for local access – No onboard backup storage – Manual Wi-Fi configuration if required – Relatively low accuracy and durability – No local data visualization graphs. – High GPS drift (more than 10 meters) if there are low satellite counts

__Table 4:__ Advantages and disadvantages of the chosen system design;

In general, the dual-access architecture enables flexibility while maintaining a low-cost, reliable, and modular design. The primary advantages and limitations are summarized in *Table 4*, highlighting the trade-offs between simplicity and functional constraints. These observations provide a clear basis for future refinement, which we will talk about in the next chapter.

CHAPTER 5: DISCUSSION AND IMPROVEMENTS

5.1 Discussion on Data Collection with URL

We incorporate two interfaces to display the air quality monitoring system data: a local web server hosted on the ESP32 and a cloud-based platform. The local website, which is accessible only to devices that connected to the same 2.4 GHz Wi-Fi network, displays real-time environmental parameters including PM1, PM2.5, PM10, temperature, relative humidity, atmospheric pressure, and GPS coordinates. Particulate matter values are presented in micrograms per cubic meter (***µg/m³***), which is more appropriate than ppm for ambient air quality monitoring as it reflects mass concentration commonly used in environmental standards. The temperature is measured in degrees Celsius (**•*C*) with an accuracy of around ±0.6 °C, while humidity is recorded as a percentage (***%***) as moisture can influence particulate behavior. We design the system to measure the atmospheric pressure in hector-pascals (***hPa***¸ converted from normal Pacal unit with `<_bme.pressure()/100.0F>`) which adds environmental reference for weather-related analysis. As mentioned, the local interface refreshes automatically every 4 seconds and the site functions reliably across different screen sizes.
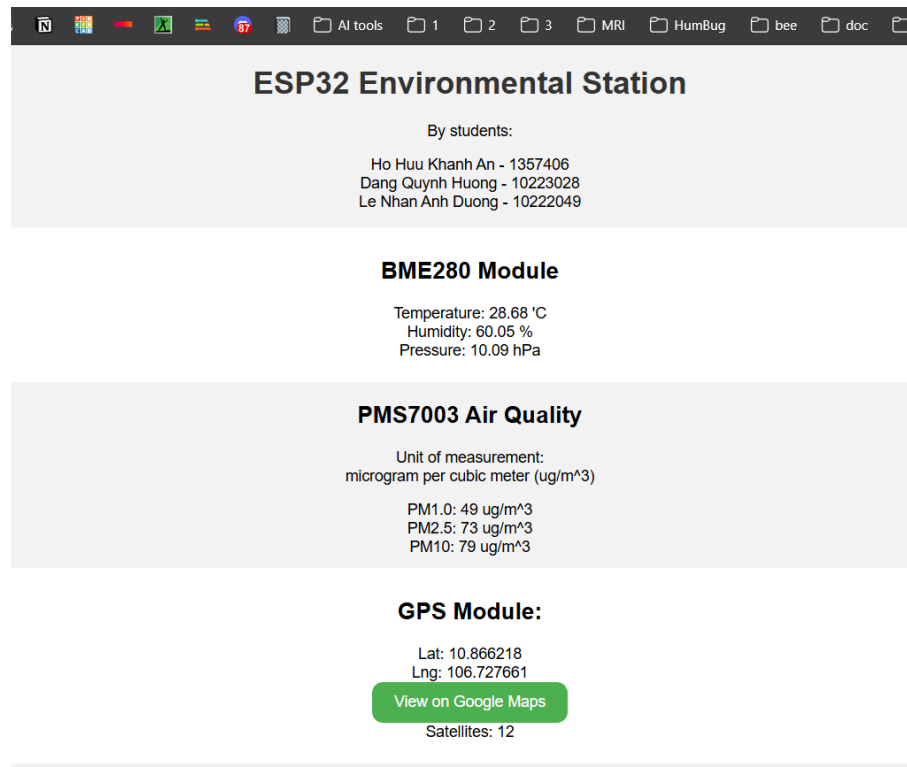

Figure 5: *Real-time data display on the ESP32 local web interface;*

For remote access, the sensor data (excluding GPS) are uploaded to *our ThingSpeak* as described in the below *Figure 6*, where they are displayed as time-series graphs with distinct color lines. The cloud platform refreshes at 15 seconds and allows exporting record for offline analysis. Sensor readings only show minor fluctuations of approximately ±6% under unchanged conditions. However, the GPS drift remains limited to a radius of 3–5 meters once satellite lock is achieved. During power interruptions, both interfaces temporarily lose recording, normal operation resumes within about 5 seconds after reboot and follows by a 7–10 seconds calibration period. While we prioritize clear and reliable data presentation, certain practical system limitations and opportunities for enhancement remain, which will be mentioned in the following section.
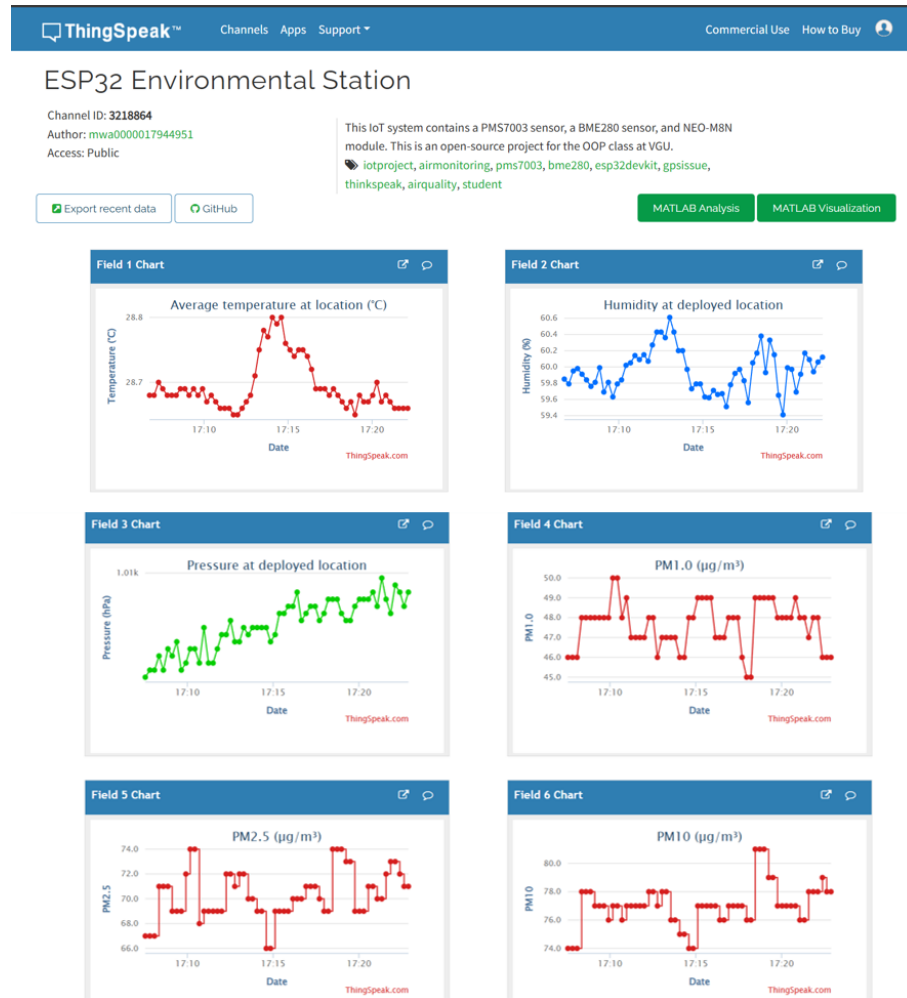

Figure 6: *Cloud-Based data visualization using ThingSpeak platform;*

5.2 Practical Constraints, Potential Improvements

Although our system operates reliably and fulfills its intended objectives, several practical and budgetary constraints affect its long-term deployment potential. The most obvious limitation is the use of the 30-pin ESP32 breakout shield. Although it is one of the few methods to allow 5V output supply, it also restricts firmware uploading through the exposed ports. Network dependency also constrains the deployment, our ESP32 supports only 2.4 GHz Wi-Fi network and requires predefined SSID credentials. Any change in the network necessitates firmware re-uploading. We conclude that public Wi-Fi without authentications cannot be utilized, and locations lacking suitable infrastructure require a temporary router, such as *TP-Link EC120-F5*. Experimental use of cloud tunneling with *Cloudflared* demonstrated that remote access to the local web is technically feasible, but it relies on an external host device and is unsuited for standalone operation.

We also found out via testing that environmental factors introduce additional limitations. GPS performance degrades during indoor or adverse weather conditions. While the most recent valid coordinates can be retained, real-time positioning cannot be guaranteed. From a data management perspective, the local web interface prioritizes simplicity and responsiveness but lacks features such as historical data visualization, user authentication, and onboard storage, long-term data retention currently depends on the ThingSpeak. Furthermore, the use of low-cost sensors introduces potential risks related to degradation over extended operation, although we confronted no such issues during testing.

Several improvements can be implemented without significantly increasing system cost. Since this is a simple IoT-based project, we suggest moving to a Raspberry Pi microcontroller module (either *Pi Zero W*, *Pi 3b* or *Pi 3b+*) to improve programming and high-accuracy data processing with Python. Adding a non-volatile storage, such as a Micro SD card module, would enable local logging during network outages. Network flexibility could improve by implementing a Wi-Fi configuration mode for the HTML interface. This would allow credential updates without recompiling programs to the processor. While our current design has demonstrated a suitable low-cost air quality monitoring station, addressing these limitations would enhance the robustness, scalability, and flexibility without compromising affordability.

CHAPTER 6: CONCLUSION

To sum up, this project has implemented a low-cost, flexible, and simple IoT environmental monitoring station capable of measuring particulate matter, atmospheric parameters, and location tracking in near real time. Our system has demonstrated effective data visualization through both local and cloud-based platforms, sufficient accuracy for trend analysis, and small-scale monitoring purposes. Despite several limitations related to connectivity, the overall architecture proved to be reliable. The results confirm that affordable embedded platforms can be effectively utilized for environmental tracking and provide a solid foundation for future system enhancements.

REFERENCE

TEXTBOOKS:

[1] Lafore, R. (2005). *Object-oriented programming in C++*, fourth edition. Sams.

[2] Asim Zulfiqar. (2024). *Hands-on ESP32 with Arduino IDE*. Packt Publishing Ltd.

[3] Ouyang, E. (2020). *Hands-On IoT: Wi-Fi and Embedded Web Development*. Erwin Ouyang.

JOURNAL & RESEARCH ARTICLES:

[4] Othman, H., Azari, R., & Guimarães, T. (2024). *Low-Cost IoT-based Indoor Air Quality Monitoring*. Technology|Architecture + Design, 8(2), 250–270. https://doi.org/10.1080/24751448.2024.2405403

[5] Pineda-Tobón, D. M., Espinosa-Bedoya, A., & Branch-Bedoya, J. W. (2024). *Aquality32: A low-cost, open-source air quality monitoring device leveraging the ESP32 and google platform*. HardwareX, 20, e00607. https://doi.org/10.1016/j.ohx.2024.e00607

[6] Siddiqui, M. A., Baig, M. H., & Yousuf, M. U. (2024). *Performance and data acquisition from low-cost air quality sensors: a comprehensive review*. Air Quality, Atmosphere & Health. https://doi.org/10.1007/s11869-024-01683-3

[7] *EFFECTIVE AIR QUALITY MONITORING SYSTEM WITH PMS7003*. (2024), TANZ (ISSN NO: 1869-7720), 19(1), https://tanzj.net/wp-content/uploads/2024/02/TJ-1480.pdf

[8] Kim, D., Shin, D., & Hwang, J. (2023). *Calibration of Low-cost Sensors for Measurement of Indoor Particulate Matter Concentrations via Laboratory/Field Evaluation*. Aerosol and Air Quality Research, 23(8), 230097. https://doi.org/10.4209/aaqr.230097

[9] Taştan, M. (2022). *A low-cost air quality monitoring system based on Internet of Things for smart homes*. Journal of Ambient Intelligence & Smart Environments, 14(5), 351–374. https://doi.org/10.3233/ais-210458

[10] Omkar, N., Rahul, S., Vijaykrishna, R., Swapna, N., & Sura, M. (2024). *Smart Environmental Monitoring Using Esp32 Microcontroller*. 19(3), 7–12. https://doi.org/10.9790/2834-1903010712

APPENDIX A — Source Code for Main Implementation

This appendix contains the source code used for the main implementation within the system's ESP32 program framework. The codes in this appendix, as well as in those following this, are written in C++ in Visual Studio Code and they uploaded to the device via PlatformIO as described earlier. Private information about Wi-Fi connectivity will be redacted. To replicate this project, please visit our **project GitHub** for further explanations.

```cpp
// file: main application file for ESP32 Air Quality Station with Web UI
#include <WiFi.h>
#include <WebServer.h>
#include <Wire.h>
#include <HTTPClient.h>
#include "WebServerUI.h"
#include "GPS.h"
#include "BME280.h"
#include "PMS7003.h"

// ---- Local 2.4GHz WiFi only !! ----
const char* ssid = " ---------- ";
const char* password = " ---------- ";
const char* THINGSPEAK_API_KEY = " ---------- ";
const char* THINGSPEAK_SERVER  = "http://api.thingspeak.com/update";
// ---- Objects ----
WebServer server(80);
HardwareSerial GPSSerial(2);
HardwareSerial PMSSerial(1);
BME280 bme;
GPS gps(GPSSerial);
PMS7003 pms(PMSSerial);
WebServerUI webUI(server, bme, gps, pms);
// main setup functions
void setup() {
  Serial.begin(115200);
// --- PMS7003 UART 1 ---
  PMSSerial.begin(9600, SERIAL_8N1, 26, 27); // using ESP pins 26 and 27
// --- BME280 I2C ---
  Wire.begin(21, 22);
  if (!bme.begin(0x76)) {
    Serial.println("✘ BME280 not found");
  } else {
    Serial.println("✅ BME280 detected");
  }
// --- GPS UART 2 ---
  GPSSerial.begin(9600, SERIAL_8N1, 16, 17); // using ESP pins 16 and 17
```

```
    Serial.println("GPS UART started");
// --- Wi-Fi connection loop ---
  WiFi.mode(WIFI_STA);
  WiFi.disconnect(true);
  delay(1000);
  WiFi.begin(ssid, password);
  Serial.print("Connecting to WiFi");
  while (WiFi.status() != WL_CONNECTED) {
    delay(500);
    Serial.print(".");
  }
// --- For Wi-Fi to work properly: ---
  unsigned long startAttemptTime = millis();
  while (WiFi.status() != WL_CONNECTED) {
  delay(500);
  Serial.print(".");
    if (millis() - startAttemptTime > 15000) {
      Serial.println("\nX WiFi FAILED, restarting..."); // Feedback timeout
      ESP.restart();
    }
  }
  Serial.println("\n✓ WiFi connected");
  Serial.print("IP address: ");
  Serial.println(WiFi.localIP());
  webUI.begin();
  server.begin();
  Serial.println("⊕ Web server started");
}
// --- Shows data on ThingSpeak: ---
void sendToThingSpeak(
    float temp, float hum,
    float pres, uint16_t pm1,
    uint16_t pm25, uint16_t pm10,
    double lat, double lng )
{
    if (WiFi.status() != WL_CONNECTED) return;
    HTTPClient http; // send data to client ThinkSpeak cloud
    String url = String(THINGSPEAK_SERVER) +
        "?api_key=" + THINGSPEAK_API_KEY +
        "&field1=" + String(temp) +
        "&field2=" + String(hum) +
        "&field3=" + String(pres) +
        "&field4=" + String(pm1) +
        "&field5=" + String(pm25) +
        "&field6=" + String(pm10) +
        "&field7=" + String(lat, 6) +
        "&field8=" + String(lng, 6);
```

```
      http.begin(url); http.GET(); http.end(); // create HTTP and start page
}
// main subsystem loop function
void loop() {
  gps.update();
  bme.update();
  pms.update();
// Send data to ThingSpeak every 15 seconds
unsigned long lastThingSpeakUpdate = 0;
  if (millis() - lastThingSpeakUpdate > 15000) {
    lastThingSpeakUpdate = millis();
    sendToThingSpeak(
        bme.temperature(),
        bme.humidity(),
        bme.pressure(),
        pms.pm1(),
        pms.pm25(),
        pms.pm10(),
        gps.latitude(),
        gps.longitude()
    );
  }
  static unsigned long lastPMS = 0; // separated delay for updating pms
  if (millis() - lastPMS >= 1000) {
    lastPMS = millis();
    if (pms.update()) {}
  }
  server.handleClient();
}
```

APPENDIX B — Source Codes for GPS Function

        This is the appendix for the implementation code and the header code of the GPS module NEO-M8N. Minor comment details are changes to better explain the program.

File name: GPS.cpp

```cpp
// file: implementation for GPS class
#include "GPS.h"
GPS::GPS(HardwareSerial& serial)
    : _serial(serial) {}
void GPS::begin(uint32_t baud) {_serial.begin(baud);}
void GPS::update() {
    while (_serial.available() > 0) {
        _gps.encode(_serial.read());
    }
}
bool GPS::hasFix() {
    return _gps.location.isValid(); } // confirm coordinates
double GPS::latitude() {
    return _gps.location.lat(); }
double GPS::longitude() {
    return _gps.location.lng(); }
uint32_t GPS::satellites() {
    return _gps.satellites.value(); } // number of satellites
```

File name: GPS.h

```cpp
// file: header for GPS class
#ifndef GPS_H
#define GPS_H
#include <TinyGPSPlus.h> // GEO-M8N Arduino Library
#include <HardwareSerial.h> // For serial logging and display
class GPS {
public:
    GPS(HardwareSerial& serial);
    void begin(uint32_t baud = 9600);
    void update();
    TinyGPSPlus& getGPSData();
    bool hasFix();
    double latitude(); double longitude();
    uint32_t satellites();
private:
    HardwareSerial& _serial;
    TinyGPSPlus _gps;
};
#endif
```

APPENDIX C — Source Codes for BME280 Function

       This is the appendix for the implementation and header program of the Bosch's humidity, atmospheric, and temperature module BME280. Only minor comment changes are made compared to the original file to better explain the program.

File name: BME280.cpp

```cpp
// file: implementation for BME280 class
#include "BME280.h"
bool BME280::begin(uint8_t address) {
    return _bme.begin(address);
}
void BME280::update() {
    _temperature = _bme.readTemperature();
    _humidity = _bme.readHumidity();
    _pressure = _bme.readPressure() / 100.0F; // convert unit from Pa to hPa
}
float BME280::temperature() const {
    return _temperature; }
float BME280::humidity() const {
    return _humidity; }
float BME280::pressure() const {
    return _pressure; }
```

File name: BME280.h

```cpp
// file: header for BME280 class
#ifndef BME280_H
#define BME280_H
#include <Adafruit_BME280.h> // Adafruit's BME280 Library
class BME280 {
public:
    bool begin(uint8_t address = 0x76);
    void update();
    float temperature() const;
    float humidity() const;
    float pressure() const;
private:
    Adafruit_BME280 _bme;
    float _temperature = 0;
    float _humidity = 0;
    float _pressure = 0;
};
#endif
```

APPENDIX D — Source Codes for PMS7003 Function

        This is the appendix for the implementation and header codes of the PMS7003 particulate sensor. Only minor comment changes are made compared to better explain the program.

File name: PMS7003.cpp

```cpp
// file: implementation for PMS7003 class
#include "PMS7003.h"
PMS7003::PMS7003(HardwareSerial& serial)
    : _serial(serial), _pms(serial) {}
void PMS7003::begin(uint32_t baud) {
    _serial.begin(baud);
}
bool PMS7003::update() {
    PMS::DATA data; // read PM1.0, PM2.5, and PM10 values
    if (_pms.read(data)) {
        _pm1_0 = data.PM_AE_UG_1_0;
        _pm2_5 = data.PM_AE_UG_2_5;
        _pm10 = data.PM_AE_UG_10_0;
        return true;    }
    return false;
}
uint16_t PMS7003::pm1() const {return _pm1_0;}
uint16_t PMS7003::pm25() const {return _pm2_5;}
uint16_t PMS7003::pm10() const {return _pm10;}
```

File name: PMS7003.h

```cpp
// file: header for PMS7003 class
#ifndef PMS7003_H
#define PMS7003_H
#include <HardwareSerial.h> // For serial logging and display
#include <PMS.h> // PMS-master Library from https://github.com/fu-hsi/PMS
class PMS7003 {
public:
    PMS7003(HardwareSerial& serial);
    void begin(uint32_t baud = 9600);
    bool update();
    uint16_t pm1() const; uint16_t pm25() const; uint16_t pm10() const;
private:
    HardwareSerial& _serial; PMS _pms;
    uint16_t _pm1_0 = 0; uint16_t _pm2_5 = 0; uint16_t _pm10 = 0;
};
#endif
```

APPENDIX E — Source Codes for Website Function

This is the appendix for the implementation and header codes of the user interface (UI) with HTML layout for the URL site 192.168.xx.xx on local network. The general UI aimed to be simple and fit to different screen's sizes. The design is only used as an extension and not as a main data graphing and recording server, which is preferred to be ThingSpeak as explained.

File name: WebServerUI.cpp

```cpp
// file: implementation for WebServerUI class
#include "WebServerUI.h"
WebServerUI::WebServerUI(WebServer& server, BME280& bme, GPS& gps, PMS7003&
pms)
: _server(server), _bme(bme), _gps(gps), _pms(pms) {}
void WebServerUI::begin() {
    _server.on("/", [this]() {
        this->handleRoot();
    });
}
void WebServerUI::handleRoot() {
    _server.send(200, "text/html", generatePage());
}
String WebServerUI::generatePage() {
// --- HTML structure ---
    String html = "<!DOCTYPE html><html><head>";
    html += "<meta charset='utf-8'>";
    html += "<meta name='viewport' content='width=device-width, initial-
scale=1'>";
    html += "<meta http-equiv='refresh' content='4'>"; // Refresh every 4 secs
    html += "<style>";
    html += "body{font-family:Arial;background:#f2f2f2;text-align:center;font-
size:14px;}";
    html += ".box{background:#fff;padding:10px 10px 20px 10px;margin:10px;
border-radius:15px;}";
    html += "h1{color:#333;}";
    html += "</style></head><body>";
    html += "<h1>ESP32 Environmental Station</h1>";
    html += "<p>By students: / redacted / </p>";
// --- BME280 ---
    html += "<div class='box'><h2>BME280 Module</h2>";
    html += "Temperature: " + String(_bme.temperature()) + " 'C<br>";
    html += "Humidity: " + String(_bme.humidity()) + " %<br>";
    html += "Pressure: " + String(_bme.pressure() / 100.0F) + " hPa</div>";
// --- PMS7003 ---
    html += "<h2>PMS7003 Air Quality</h2>";
    html += "<p>Unit of measurement: microgram per cubic meter (ug/m^3)</p>";
```

```
    html += "PM1.0: " + String(_pms.pm1()) + " ug/m^3<br>";
    html += "PM2.5: " + String(_pms.pm25()) + " ug/m^3<br>";
    html += "PM10: " + String(_pms.pm10()) + " ug/m^3<br>";
// --- GPS NEO-M8N ---
    html += "<div class='box'><h2>GPS Module:</h2>";
    if (_gps.hasFix()) {
        html += "Lat: " + String(_gps.latitude(), 6) + "<br>";
        html += "Lng: " + String(_gps.longitude(), 6) + "<br>";

        html += "<a href='https://www.google.com/maps?q="
        + String(_gps.latitude(), 6) + ","
        + String(_gps.longitude(), 6) +
        "' target='_blank'>"
        "<br><button style='background-color:#4CAF50;color:white;font-
size:14px;padding:10px 20px;border:none;border-radius:10px;cursor:pointer;'>
View on Google Maps</button></a><br>";
    } else {
        html += "<br>Waiting for GPS coordinates fix...<br>";       }
    html += "Satellites: " + String(_gps.satellites()) + "</div>";
// --- End ---
    html += "</body></html>";
    return html;
}
```

File name: WebServerUI.h

```
// file: header for WebServerUI class
#ifndef WEB_SERVER_UI_H
#define WEB_SERVER_UI_H
#include <WebServer.h> // Standard Web Access Library
#include "BME280.h"
#include "GPS.h"
#include "PMS7003.h"
class WebServerUI {
public:
    WebServerUI(
        WebServer& server, BME280& bme,
        GPS& gps, PMS7003& pms );
    void begin();
private:
    WebServer& _server; BME280& _bme;
    GPS& _gps; PMS7003& _pms;
    String generatePage(); void handleRoot();
};
#endif
```