**ACKNOWLEDGEMENT**

The successful completion of any task would be incomplete without mention of the people who made it possible through their guidance and encouragement crowns all the efforts with success. We take this opportunity to acknowledge with thanks and deep sense of gratitude to Mrs. P. CHAITANYA Assistant Professor for her constant encouragement and valuable guidance during this work. A Special note of Thanks to Associate PROF. CHINNAM S V MARUTHI RAO Associate Professor, who has been a source of Continuous motivation and support in the completion of this project. He had taken time and effort to guide and correct us all through the span of this work. We owe very much to the Department Faculty, Principal and the Management who made our term at SREYAS a Stepping stone for my career. We treasure every moment we had spent in the college. Last but not least, our heartiest gratitude to our parents and friends for their continuous encouragement and blessings. Without their support this work would not have been possible.

**MAHI SRIJA (22VE1A0492)**

**A.ALEKHYA (22VE1A0470)**

**S.GANESH (22VE1A04B8)**

TABLE OF CONTENTS

ABSTRACT i

LIST OF FIGURES ii

Chapter 1: INTRODUCTION 1 - 7

1.1 Introduction 1

1.2 Motivation 2

1.3 Objective of Project 4

1.4 Scope of the Project 5

1.5 Organization of the Thesis 7

1.6 Summary 7

Chapter 2: LITERATURE REVIEW/EXISTING SYSTEM 8 - 12

2.1 Introduction 8

2.2 Existing Air Pollution Monitoring Systems 8

2.3 Related Work in IoT-Based Monitoring 9

2.4 Limitations of Existing Solutions 10

2.5 Need for the Proposed System 10

2.6 Summary 12

Chapter 3: SYSTEM ARCHITECTURE AND DESIGN 13 - 16

3.1 System Overview 13

3.2 Functional Block Diagram 13

3.3 Data Flow and Control Flow Design 15

3.4 Hardware and Software Integration 15

3.5 Summary 16

Chapter 4: HARDWARE IMPLEMENTATION **17** - 25

4.1 Components Used 17

4.2 Circuit Design and Schematics 21

4.3 Power Supply and Layout Considerations 23

4.4 Summary 25

Chapter 5: SOFTWARE IMPLEMENTATION 26 - 45

5.1 Development environment setup 26

5.2 Wi-Fi Connectivity and Network Configuration 28

5.3 Cloud Integration: FireBase Real Time Database 31

5.3.1 FireBase project setup 31

5.3.2 Data Logging Logic 34

5.3.3 FireBase Ordino Client Library 37

5.4 Local Data Display: LCD Interface 42

5.5 Summary 45

Chapter 6: TESTING AND RESULTS **46** - 50

6.1 Test Environment and Procedure 46

6.2 Functional Testing (Scanning Accuracy) 47

6.3 Sample Output Snapshots 48

6.4 Summary 50

Chapter 7: CONCLUSION AND FUTURE SCOPE **5**1 - 54

7.1 Conclusion 51

7.2 Limitations 52

7.3 Future Scope 53

7.4 Summary 54

REFERENCES 55

APPENDIX: FULL ANNOTATED SOURCE CODE 56 - 60

I

**ABSTRACT**

This project presents a real-time loT-based air pollution monitoring system designed to detect multiple harmful gases and report data to the cloud. It uses a combination of gas sensors-MQ-135, MQ-2, and MQ-6 to measure pollutants such as carbon dioxide (CO₂), carbon monoxide (CO), ammonia (NH3), methane (CH4), and other volatile organic compounds (VOCs) in the environment. These sensors are interfaced with a Wi-Fi-enabled microcontroller (ESP32), which reads the data and uploads it to a cloud platform such as Firebase real-time database. Users can access real-time pollution levels and historical trends through a mobile or web dashboard. The system can issue alerts when any pollutant exceeds the defined safe limits. It is ideal for indoor environments, urban monitoring, and public health applications.

II

List of Figures

Pg.No

Figure 3.1 : Block Diagram 14

Figure 4.1 : ESP32 17

Figure 4.2 : MQ-135 Sensor 18

Figure 4.3 : MQ-2 Sensor 18

**Figure 4.4** : MQ-6 Sensor 19

**Figure 4.5** : DHT-11 Sensor 19

**Figure 4.6** : Buzzer 20

**Figure 4.7** : I2C-Integrated circuit 20

**Figure 4.8 :** LCD  21

**Figure 4.9 :** BUCK Converter  21

**Figure 4.10 :** Circuit diagram 23

Figure 6.1 : Simulation Result 48

Figure 6.2 : Simulation Result 49

## CHAPTER 1

## INTRODUCTION

**1.1 Introduction**

Air pollution poses a significant threat to environmental and human health in the modern world. Rapid industrialization, urbanization, and increased vehicular emissions have led to the degradation of air quality, especially in urban areas. The traditional methods of monitoring air quality involve manual data collection and are often centralized, expensive, and not accessible in real-time. In response to these limitations, this project proposes an **IoT-based real-time air pollution monitoring system** using modern microcontroller and sensor technologies integrated with cloud platforms.

The core of the system is the **ESP32 microcontroller**, which is chosen for its dual-core performance, built-in Wi-Fi and Bluetooth, and low power consumption. The system is designed to detect the concentration of various gases using **gas sensors** like **MQ135**, **MQ2**, and **MQ6**, which are capable of sensing harmful gases such as CO₂, CO, NH₃, benzene, alcohol, smoke, and LPG. Additionally, **temperature and humidity** are monitored using the **DHT11 sensor**, which helps in correlating atmospheric conditions with pollutant behavior.The ESP32 collects data from all sensors and transmits it via Wi-Fi to a **Firebase Realtime Database**, where it is stored, updated, and visualized in real-time. Firebase allows seamless cloud integration and ensures that the data is remotely accessible from anywhere in the world using web or mobile applications. The data can be used for alerts, analytics, or long-term environmental monitoring.

An **I2C-based LCD display** is used to provide instant feedback locally by showing gas concentrations and environmental data. A **buzzer** is included as an alert mechanism to warn users when gas levels exceed safe thresholds. A **buck converter** is integrated into the power supply system to regulate voltage for stable operation of ESP32 and peripheral components. It stands as a practical and scalable model for addressing the growing challenges of pollution through smart technology.This project demonstrates the synergy of IoT and cloud computing in developing **low-cost, scalable, and portable environmental monitoring solutions**. It is ideal for deployment in schools, homes, industrial areas, and smart cities to help monitor and reduce the harmful effects of pollution.

**1.2 Motivation**

Air pollution is one of the most pressing environmental challenges faced by modern society, contributing to serious health problems such as respiratory diseases, cardiovascular issues, and premature deaths. In densely populated urban areas, pollution levels often rise unnoticed due to the lack of real-time and localized monitoring systems. Conventional air quality monitoring setups are typically large-scale, expensive, and limited to government or institutional use, making them inaccessible for everyday public awareness or community-based actions. This gap in accessibility and responsiveness inspired the development of a low-cost, real-time, and IoT-enabled air pollution monitoring system that can be deployed at a micro level — such as in homes, schools, and neighborhoods. The motivation behind this project is to empower individuals and communities with technology that not only monitors air quality effectively but also provides instant feedback and remote access through cloud platforms. By integrating sensors, microcontrollers, cloud databases, and alert mechanisms, the system aims to promote environmental consciousness, support preventive health care, and contribute to smarter, cleaner living environments.

**Health Impact and Lack of Public Awareness**  
Long-term exposure to polluted air can result in chronic respiratory issues, heart disease, asthma, and even cancer. Many people remain unaware of the poor air quality they are exposed to on a daily basis because traditional monitoring stations are few, far between, and do not provide individual-level data. This project is motivated by the need to raise awareness and encourage informed decisions through localized monitoring.

**Advancement in IoT and Sensor Technologies**  
The emergence of powerful microcontrollers like the ESP32, combined with low-cost gas sensors (MQ series) and cloud services (like Firebase), has made it possible to design and deploy smart, wireless, and scalable monitoring devices. This technological advancement is a key driver behind the feasibility and motivation of this project.

**Need for Real-Time and Remote Monitoring**  
In emergency situations like gas leaks or industrial emissions, timely alerts can save lives. Traditional systems lack immediate responsiveness. This project aims to implement real-time sensing with instant feedback via a buzzer and cloud-based dashboards accessible from any location, enhancing both safety and control.

**Educational and Research Motivation**  
From an academic and research perspective, this project serves as a practical implementation of interdisciplinary knowledge involving IoT, embedded systems, cloud computing, and environmental science. It motivates students and researchers to explore how technology can be applied to solve real-world problems

**1.3 Objective of Project**

The objective of this project is to design and implement a real-time IoT-based air pollution monitoring system that can accurately detect and measure harmful gases such as CO₂, CO, NH₃, CH₄, and VOCs using multiple gas sensors (MQ-135, MQ-7, MQ-6). The system aims to transmit environmental data to a cloud platform via a Wi-Fi-enabled microcontroller (ESP32), enabling users to monitor pollution levels remotely through a mobile or web dashboard. The system will also be capable of generating alerts when pollutant levels exceed safe limits, making it suitable for indoor environments, urban areas, and public health applications.

****To develop a real-time air quality monitoring system using ESP32 microcontroller :**** Utilize the ESP32 for its built-in Wi-Fi and processing capabilities to collect data from multiple environmental sensors and transmit it wirelessly to the cloud.

****To detect and measure harmful gases using a combination of gas sensors (MQ135, MQ2, MQ6):**** Accurately sense pollutant concentrations such as CO₂, NH₃, benzene, LPG, smoke, and other toxic gases, ensuring comprehensive air quality data.

****To measure environmental parameters like temperature and humidity using DHT11 sensor:**** Integrate the DHT11 sensor to enhance pollution analysis by correlating environmental factors that may influence gas sensor readings.

****To store and display sensor data on a cloud-based platform (Firebase Realtime Database):**** Implement cloud connectivity for data logging, real-time updates, and remote access, using Firebase as a lightweight and scalable backend service.

****To provide a local display interface using an I2C-enabled LCD screen:**** Offer immediate and on-site data visualization through an LCD, showing key parameters such as gas levels, temperature, and humidity.

****To alert users with a buzzer when pollution levels exceed predefined thresholds:**** Add an auditory alert mechanism that can provide instant warnings in hazardous conditions.

**1.4 Scope of the Project**

The scope of this project is carefully defined to ensure the development of a functional and robust prototype while acknowledging areas for future expansion.

**The project's scope includes:**

1. **Hardware Requirements**

The hardware includes the **ESP32 microcontroller**, chosen for its built-in Wi-Fi and efficient performance. **MQ135**, **MQ2**, and **MQ6** gas sensors detect harmful gases like CO₂, NH₃, LPG, and smoke. The **DHT11** sensor measures temperature and humidity. An **I2C-based LCD** displays real-time values, and a **buzzer** alerts users when pollutant levels exceed thresholds. A **buck converter** ensures regulated voltage for all components. These are all integrated on a breadboard or PCB for compactness and portability.

**2. Software Requirements**

The system is programmed using the **Arduino IDE**, with libraries like WiFi.h, DHT.h, FirebaseESP32.h, and LiquidCrystal\_I2C.h to manage sensor data, cloud connectivity, and display control. The **Firebase Realtime Database** is used to store and retrieve data remotely. Optionally, mobile or web apps can be built using **MIT App Inventor** or **HTML/CSS** to present the data in a user-friendly way.

**3. Core Functionality**

The system continuously monitors gases and environmental conditions through sensors. Data is processed by the ESP32, shown on the LCD, and sent to the Firebase cloud. When pollutant levels are high, a buzzer gives an immediate alert. The user can access live and historical data from anywhere, making it a complete real-time monitoring and alerting system.

**4. Data Presentation**

Sensor readings are shown locally on a **16x2 I2C LCD**, while also being uploaded to **Firebase** for remote access. Data can be viewed in real time on a dashboard or mobile app, allowing users to track trends, get alerts, and take quick action if needed. Firebase also allows time-stamped data storage for historical analysis.

**5. System Architecture**

The system is structured into three layers: **sensing**, **processing**, and **cloud communication**. Sensors collect environmental data, the **ESP32 processes it**, displays it on the LCD, and triggers alerts if needed. It also pushes data to **Firebase** over Wi-Fi.

**1.5 Organization of the Thesis**

This document is organized into eight chapters. The initial chapters introduce the project and its core technologies. The subsequent chapters cover the system architecture, hardware implementation, software development, and web interface design. The final chapters present the testing methodology and results, followed by a conclusion and discussion of future work. The document concludes with references and an appendix containing the annotated source code.

**1.6 Summary**

This project presents the development of an IoT-based air pollution monitoring system that provides real-time sensing, alerting, and data sharing using the ESP32 microcontroller. The system integrates multiple gas sensors (MQ135, MQ2, MQ6) to detect harmful pollutants, a DHT11 sensor for measuring temperature and humidity, and an I2C LCD to locally display readings. A buzzer is included to provide alerts when pollution levels exceed safe thresholds. The ESP32 wirelessly transmits the collected data to the **Firebase Realtime Database**, enabling remote access via mobile or web platforms. A buck converter ensures proper power supply to the components. The system is compact, low-cost, and easily deployable, making it suitable for use in homes, schools, industries, and public areas. With its cloud-based design and modular structure, the project supports scalability and future upgrades like mobile apps, GPS, and data analytics. This smart monitoring system promotes awareness, safety, and sustainable living by empowering users to monitor air quality anytime, anywhere.

## CHAPTER 2

**LITERATURE REVIEW/EXISTING SYSTEM**

**2.1 Introduction to IOT**

Air pollution is a serious environmental issue that affects the health and well-being of populations across the globe. With the rapid industrialization and urbanization in recent decades, the levels of toxic gases and particulate matter in the air have increased significantly. Traditional air pollution monitoring systems, while accurate, are often large, expensive, and not accessible to the general public. This chapter explores existing systems, previously conducted research, current IoT-based solutions, their limitations, and identifies the need for a low-cost, real-time, cloud-connected air monitoring solution like the one proposed in this project. A thorough literature review helps us understand where current technologies stand, what innovations are already in use, and what gaps remain unaddressed.

### **2.2** Existing Air Pollution Monitoring Systems

Government agencies and research institutions have developed several large-scale air quality monitoring systems that are deployed in urban areas. These include:

**Fixed Monitoring Stations:** Installed by national pollution control boards (e.g., CPCB in India), these stations measure a wide range of air pollutants such as CO, NOx, SOx, PM2.5, and PM10 using high-precision instruments. However, they are limited in number and mainly placed in urban centers.

**Satellite-Based Monitoring:** Agencies like NASA provide global data on air quality through satellite imagery, which can be useful for macro-level analysis but lacks real-time granularity at the local level.

**Mobile Van Stations:** Equipped with high-end sensors and analyzers, these are used for mobile assessments but are costly and logistically complex to manage.

While these systems provide accurate data, they are inaccessible to the general population for real-time awareness. Additionally, the infrastructure is too expensive and large-scale for deployment in homes or small-scale environments.

### **2.3** Related Work in IoT-Based Monitoring

With the rise of the Internet of Things (IoT), several researchers and hobbyists have proposed or implemented small-scale air quality monitoring systems. Some of the notable approaches include:

**Arduino and GSM-Based Systems:** Used for SMS-based alerts and periodic data reporting, but lacking real-time cloud visualization.

**Raspberry Pi with Web Servers:** Capable of running Python scripts to monitor air quality and upload data to web servers. However, Raspberry Pi setups are more power-hungry and expensive.

**NodeMCU/ESP8266 Projects:** Often limited to a few sensors due to memory and GPIO constraints.

**Firebase-Based Weather Stations:** Implemented for tracking temperature and humidity, these projects have shown the ease of integrating Firebase with microcontrollers.

While these projects have paved the way for DIY environmental monitoring, they are often limited in scope, not optimized for multi-sensor setups, and lack both local alert mechanisms and full integration of display + cloud + alert features.

### **2.4** Limitations of Existing Solutions

Despite the progress made in IoT and cloud computing, current solutions still exhibit several limitations:

**Cost:** Many systems use expensive boards or sensors not feasible for mass or community-level deployment.

**Accessibility:** Most systems do not offer simple, user-friendly interfaces for non-technical users.

**Alert Mechanisms:** Few solutions include real-time alerts (buzzer/LEDs) when pollution levels spike.

**Cloud Dependence:** Some are not cloud-integrated, or the cloud setup is too complex for small projects.

**Scalability:** Existing solutions are not modular or easy to expand with new features.

### **2.5** Need for the Proposed System

To address the above limitations, there is a need for a low-cost, compact, and user-friendly air pollution monitoring system that

Uses **ESP32**, which is affordable, efficient, and comes with built-in Wi-Fi, enabling direct cloud connectivity.

Integrates **multiple gas sensors (MQ135, MQ2, MQ6)** to detect a wide range of common air pollutants including CO2, CO, NH3, LPG, and smoke.

Monitors **temperature and humidity** using the **DHT11** sensor, providing additional environmental context.

Uploads data to the **Firebase Realtime Database**, offering seamless and scalable data access via the internet.

Provides **local display** using an **I2C LCD** and **immediate alerts** through a **buzzer** for enhanced usability.

Incorporates a **buck converter** to stabilize and regulate voltage supply, ensuring consistent performance.

Is **scalable and modular**, allowing future integration of features such as GPS for location tagging, mobile app support, or AI models for predictive analytics.

The proposed system offers a practical, real-time, cloud-connected solution that can be easily deployed in urban and rural settings to monitor air quality and promote health awareness.

**2.6 Summary**

This chapter reviewed various traditional and modern air pollution monitoring systems, highlighting their scope and limitations. While existing systems are effective at large scale, they are costly, inaccessible, and complex. Current IoT-based solutions provide some improvements but still lack full integration and usability. The proposed project bridges this gap by offering a low-cost, real-time, and cloud-enabled solution that combines sensing, display, alerting, and data storage in one efficient system. This review justifies the need for a smart, accessible, and scalable air pollution monitoring solution based on IoT technologies.

## CHAPTER 3

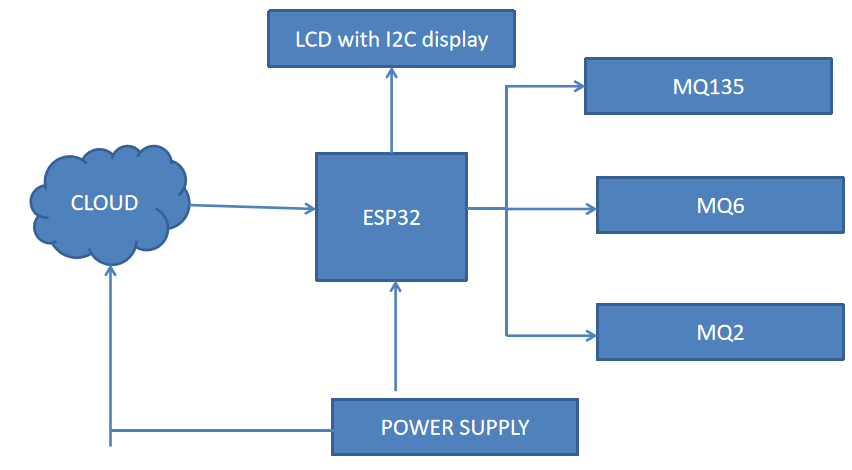
**SYSTEM ARCHITECTURE AND DESIGN**

**3.1 System Overview**

The IoT-based air pollution monitoring system is structured into layered modules to ensure smooth operation, scalability, and ease of integration. At its core is the ESP32 microcontroller, which orchestrates the collection, processing, and transmission of environmental data. The sensor layer includes MQ135, MQ2, and MQ6 gas sensors for detecting various air pollutants, as well as the DHT11 sensor for monitoring temperature and humidity. These sensors feed real-time data to the ESP32, which applies decision logic based on pre-set thresholds. Alerts are conveyed locally through a buzzer and visually via an I2C-connected LCD. The ESP32 also connects to a Firebase Realtime Database, enabling remote data storage and access through the cloud. This architecture facilitates both local alerts and global monitoring. The modular design ensures easy updates and expansions. Each layer—sensing, processing, output, cloud, and power—is optimized for reliability and low energy consumption. The buck converter ensures a regulated power supply, maintaining hardware safety. This layered and interconnected setup allows continuous monitoring, making it suitable for deployment in diverse environmental conditions.

**3.2 Functional Block Diagram**

The functional block diagram provides a graphical representation of the core modules and their interconnections in the system. It begins with the input layer, comprising MQ135, MQ2, MQ6, and DHT11 sensors. These feed environmental data into the ESP32, the central processing unit. The ESP32 processes this input data using pre-defined logic and performs actions such as displaying the values on an LCD and activating a buzzer in case thresholds are exceeded. Concurrently, it establishes a Wi-Fi connection to upload sensor data to the Firebase Realtime Database. The cloud storage allows for data retrieval, logging, and remote analysis. Power is supplied through a buck converter, ensuring all components receive stable voltages. Arrows in the diagram indicate the data flow, starting from sensors, passing through ESP32, and ending at the output and cloud modules. The diagram helps to quickly understand the system’s workflow, interaction, and fault points. It is also essential for hardware debugging and software development. Each module is represented with standard symbols for clarity. This visual structure facilitates collaboration, documentation, and future enhancements.



**Fig. 3.1 : Block diagram**

**3.3 Data Flow and Control Flow Design**

The data flow design of the system outlines the path and transformation of data from sensors to cloud and local display. Initially, raw analog/digital signals are captured by the gas and temperature/humidity sensors. These are then sent to the ESP32, which digitizes and processes the readings. The ESP32 applies threshold-based logic to determine whether the values indicate normal or hazardous air quality. Based on this evaluation, corresponding outputs are triggered—LCD updates, buzzer alarms, and cloud uploads. The control flow governs the decision-making logic of the ESP32. It initializes modules, manages retries in case of failures, and ensures that each operation completes before the next starts. Timers and flags are used to control sensor reading intervals and data upload frequency. Fail-safe conditions are embedded to prevent incorrect operations due to sensor or Wi-Fi issues. The design prioritizes efficiency and reliability, ensuring no data is lost during transmission. Both data and control flows operate concurrently, managed via loop functions and interrupts in the firmware. This combined flow structure enables real-time responsiveness and system robustness.

.

**3.4 Hardware and Software Integration**

Hardware and software are tightly integrated in this system to ensure a seamless operation. The ESP32 microcontroller acts as the bridge between the physical world (sensors and actuators) and the digital world (Firebase and display outputs). On the hardware side, sensors are connected to specific GPIO and analog pins, with the I2C LCD using two-wire communication to save pins. The buck converter provides stable voltage, preventing hardware failures. On the software side, the Arduino IDE is used to develop and upload the program that governs the entire system’s behavior. The program initializes sensor drivers, sets up Wi-Fi and Firebase connections, reads data periodically, and implements the logic for thresholds and outputs. Libraries such as WiFi.h, FirebaseESP32.h, and LiquidCrystal\_I2C.h simplify communication with external modules. The software also includes serial monitor outputs for debugging and uses modular code to allow easy updates. Error handling routines ensure system stability. Together, the hardware-software interaction ensures real-time, accurate, and reliable performance.

**3.5 Summary**

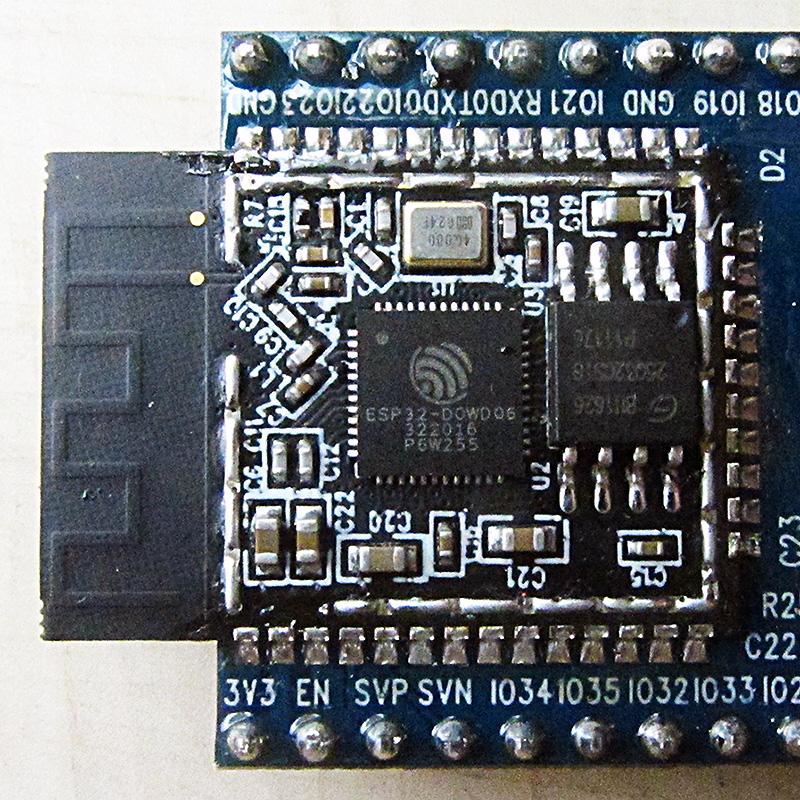
This chapter detailed the architecture and design of the IoT-based air pollution monitoring system. Beginning with an overview, we examined the system’s modular layout comprising sensor, processing, output, power, and cloud layers. The functional block diagram clarified the interaction between these modules. Data and control flow designs explained how information is acquired, processed, and acted upon in real-time. Hardware and software integration was discussed in terms of pin connections, power regulation, code structure, and logic implementation. This cohesive design ensures the system’s reliability, scalability, and efficiency. It supports continuous monitoring and real-time alerts for varied environmental conditions. The chapter establishes a clear foundation for implementation, testing, and further development in the next stages of the project.

**Chapter 4**

**HARDWARE IMPLEMENTATION**

**4.1 Components Used**

The hardware design for this project is intentionally minimalist, relying on two primary, highly-integrated components. This approach reduces complexity and cost while still delivering powerful functionality.

* **ESP32 Development Board** The central component of the system is a standard ESP32 development board (such as an ESP32 DevKitC). This board serves as the brain of the operation, providing the processing power, Wi-Fi radio, and necessary I/O pins. The board is powered via a 5V micro-USB connection.

**Fig. 4.1: ESP32**

### MQ135 – Air Quality Sensor

The MQ135 detects harmful gases like ammonia, CO₂, benzene, and smoke.It works by changing its resistance based on gas concentration.Outputs an analog voltage that the ESP32 reads to assess air quality.Requires 5V power and has a preheat time for accurate readings.Widely used for indoor air monitoring due to its high sensitivity.  
Helps trigger alerts when pollution crosses safe levels.



**Fig. 4.2 : MQ-135 Sensor**

### MQ2 – Smoke and Gas Sensor

The MQ2 senses smoke, LPG, methane, hydrogen, and alcohol.It changes resistance when exposed to combustible gases.Provides both analog and digital outputs for flexible use.Requires a small warm-up period to stabilize readings.Commonly used in gas leak detection systems.Offers quick response time for safety-critical environments.



**Fig. 4.3 : MQ-2 Sensor**

### MQ6 – LPG and Flammable Gas Sensor

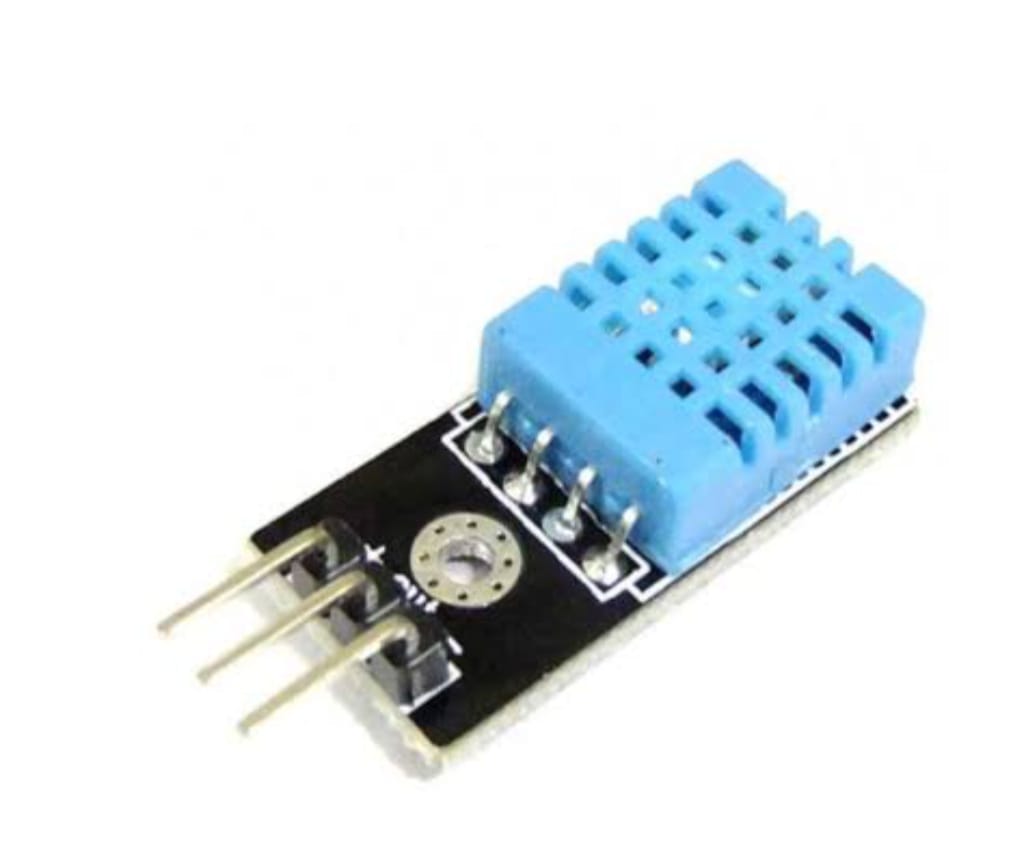
The MQ6 is sensitive to LPG, butane, propane, and similar gases.It detects flammable gases through resistance change in its sensor.Offers analog and threshold-based digital outputs.Operates at 5V and has a fast response time.Used in homes and industry to detect gas leaks.It enhances safety by enabling early leak detection

****

**Fig. 4.4 : MQ-6 Sensor**

### DHT11 – Temperature and Humidity Sensor

The DHT11 measures ambient temperature and humidity.It provides digital output and operates at 3.3V–5V.Communicates with the ESP32 via a single-wire protocol.Accuracy is ±2°C for temperature and ±5% for humidity.Has a sampling rate of 1 reading per second.Useful in correlating environmental conditions with air quality.

****

**Fig. 4.5 : DHT-11 Sensor**

### Buzzer – Alert Module

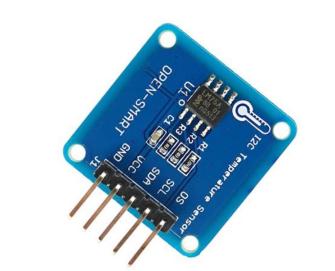
The buzzer provides an **audible alarm** when pollution is high.It is controlled by a digital signal from the **ESP32**.Works on 3.3V or 5V and is low power.Can emit continuous or pulse beeps as alerts.Adds real-time feedback to users without a display.Used to quickly alert users in case of danger or threshold breach.

****

**Fig. 4.6 : Buzzer**

### I2C – Inter-Integrated Circuit

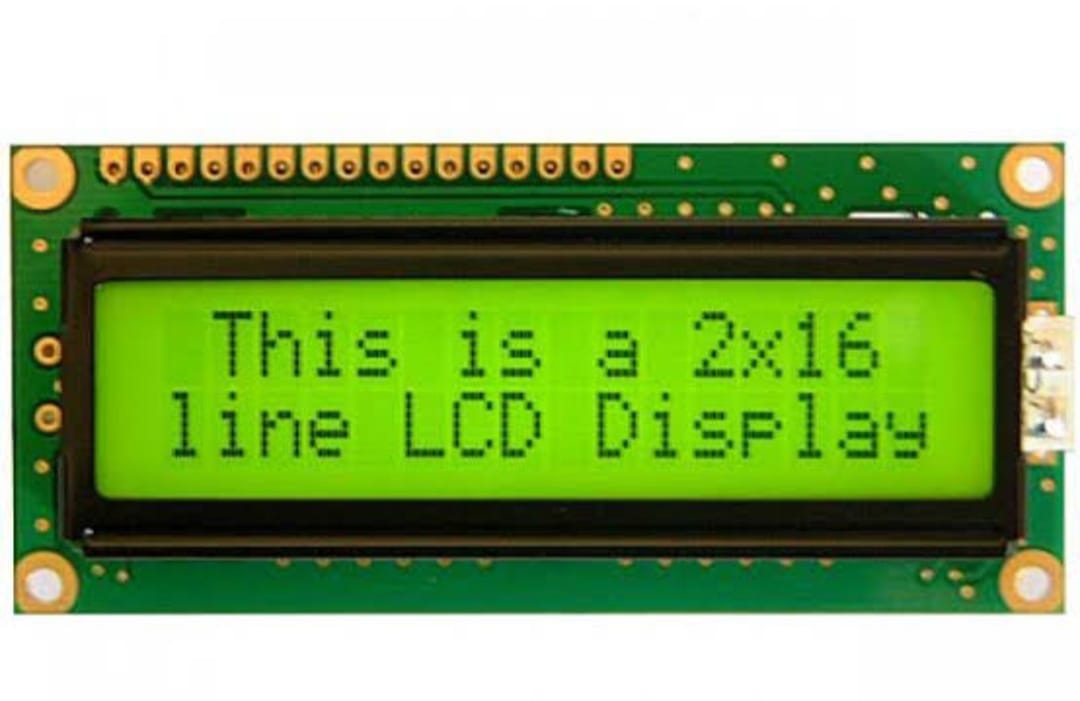
**I2C** is a two-wire serial communication protocol (SDA & SCL).It allows multiple devices to communicate with the ESP32.Used in this project to interface the **LCD display**.It reduces the number of GPIO pins needed.Supports long-distance communication between ICs.Essential for scalable and clean wiring in embedded systems.

****

**Fig. 4.7 : I2C-Integrated Circuit**

### LCD (I2C 16x2 Display)

The **16x2 LCD** shows real-time sensor readings like gas level.Using **I2C**, only two pins (SDA, SCL) are needed.Powered by 5V and easily controlled via a library.It improves user interaction with live feedback.Can display text like “Air Quality: GOOD” or numeric values.Compact and readable, ideal for small embedded systems.

****

**Fig. 4.8 : LCD**

### Buck Converter – Voltage Regulator

A buck converter steps down high voltage to a stable lower level.Used to convert 12V or 9V to **5V or 3.3V** for safe ESP32 use.Prevents over-voltage damage to sensors and controller.Operates efficiently with low heat loss.Essential for battery or adapter-powered systems.Ensures consistent power supply for reliable operation.



**Fig. 4.9 : Buck Converter**

**4.2 Circuit Design and Schematics**

The image showcases a well-organized prototype assembled on a breadboard. At the heart of the setup is the ESP32 microcontroller mounted on the bottom-right section of the board, responsible for collecting data from sensors, triggering outputs, and uploading values to the cloud.

Sensors Used: MQ135, MQ2, and MQ6 gas sensors are located at the top (left to right) and connected to analog input pins of the ESP32. These sensors detect different types of gases such as CO, CO₂, smoke, and LPG.

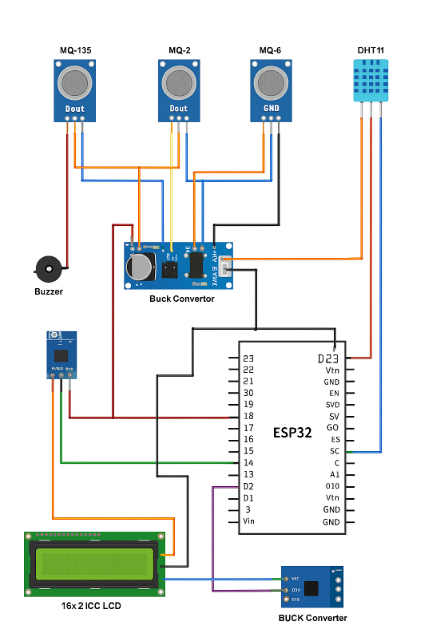
DHT11 sensor (far-right) is connected using a digital data line for temperature and humidity monitoring.

Output Devices:

I2C 16x2 LCD is mounted at the bottom-left, connected via an I2C module that communicates using SDA (blue wire) and SCL (purple wire).A buzzer (center-top, black component) is connected to a GPIO pin, used to alert when pollutant levels cross a threshold.

Power Supply: A buck converter is visible above the ESP32, stepping down voltage (likely from 12V to 5V or 3.3V), ensuring stable power to the ESP32 and peripherals.Power and GND wires are color-coded (red for VCC, black/white for GND), with all components sharing a common ground.

Wiring & Integration: Connections are made with jumper wires and zip-tied for neatness.The I2C module reduces pin usage on the ESP32.The breadboard offers a semi-permanent layout while retaining flexibility for modifications.



**Fig. 4.10 : Cicuit diagram**

**4.3 Power Supply and Layout Considerations**

**Power Supply**: The power supply system in an IoT-based air pollution monitoring project ensures stable and efficient operation of all components. It begins with an external DC source, typically a 9V or 12V battery or adapter, which serves as the primary power input. This voltage is too high for the ESP32 and other modules, so a buck converter is employed to step down the voltage to a stable 5V. The buck converter is a DC-DC converter known for its efficiency and minimal heat loss compared to linear regulators. This 5V output is distributed to the gas sensors (MQ2, MQ6, MQ135), DHT11, buzzer, and the I2C LCD display.

The ESP32, which operates on 3.3V logic levels, has an onboard voltage regulator that further drops the 5V supply down to 3.3V for its internal use. The entire power line is arranged carefully to avoid voltage drops and noise. Decoupling capacitors are often used across power lines to smoothen fluctuations. Proper grounding is crucial and a common ground line connects all modules. In battery-powered versions, power efficiency becomes essential, so sleep modes and efficient code help reduce consumption.

Protection mechanisms like reverse polarity protection and fuse-based over current protection can be added to safeguard the circuit. The use of the buck converter ensures longevity, energy efficiency, and safe operation across all hardware. This makes the power subsystem a critical part of the project’s stability and reliability

**Layout and Assembly**: The **layout and assembly** of the IoT-based air pollution monitoring system are structured to ensure clarity, stability, and ease of troubleshooting. The components are mounted on a **general-purpose breadboard**, offering a semi-permanent setup ideal for prototyping. The **ESP32 microcontroller** is placed centrally at the bottom right, making it accessible for wiring both sensors and output devices. On the top section of the board, the **MQ135**, **MQ2**, and **MQ6** gas sensors are evenly spaced to reduce interference and are connected via jumper wires to analog input pins on the ESP32.The **DHT11 sensor** is connected through a digital pin, and its wires are neatly routed with zip ties. The **I2C LCD module** is positioned at the bottom left, close to the I2C interface pins (SDA/SCL) of the ESP32, minimizing signal loss. The **buzzer** is centrally located for short connection lines and efficient sound propagation. The **buck converter**, responsible for power regulation, is installed above the ESP32 and wired to distribute 5V power to all modules.Color-coded wires (red for VCC, black/white for GND, others for data) aid visual identification.

**4.5 Summary**

This IoT-based air pollution monitoring system is designed to detect and report environmental air quality using the ESP32 microcontroller. It integrates multiple gas sensors—MQ135, MQ2, and MQ6—for detecting harmful gases such as carbon monoxide, smoke, alcohol, and ammonia. The DHT11 sensor monitors ambient temperature and humidity, which are also critical environmental factors. All sensor data is collected and processed by the ESP32, which displays real-time readings on an I2C 16x2 LCD and activates a buzzer when air quality exceeds safe limits.The system is connected to the internet using the ESP32’s built-in Wi-Fi capability, enabling it to send data to the Firebase Realtime Database. This allows users to monitor pollution levels remotely through cloud storage. A buck converter is used to step down the input voltage, ensuring safe and stable power delivery to all components.

All components are mounted on a breadboard with organized wiring and a common ground to ensure signal stability. The modular architecture makes it easy to upgrade or expand the system. This project serves as a low-cost, reliable solution for continuous air quality monitoring, ideal for use in homes, schools, and urban areas. It combines embedded systems and cloud technologies to promote environmental health awareness.

**Chapter 5**

**SOFTWARE IMPLEMENTATION**

The software implementation forms the intelligence of the IoT-based air pollution monitoring system, enabling data acquisition, processing, communication, and presentation. This section details the development environment, network connectivity, cloud integration, and local display functionalities.

### 5.1. Development Environment Setup

The Arduino IDE is chosen as the primary development environment due to its user-friendliness, extensive library support, and broad community. To program the ESP32 microcontroller within this environment, specific configurations and libraries are required.

1. Arduino IDE Installation:The first step involves installing the Arduino IDE on the development machine. While newer versions (e.g., Arduino 2.x) are available, some plugins or libraries might have better compatibility with older versions (e.g., Arduino 1.8.X).60 It is recommended to use a stable version that supports the necessary ESP32 add-ons and libraries.

2. ESP32 Board Manager Installation:To enable programming for ESP32 boards within the Arduino IDE, the ESP32 board package must be installed via the Boards Manager:

* Open the Arduino IDE.
* Navigate to File > Preferences.
* In the "Additional Board Manager URLs" field, add the following URL:https://espressif.github.io/arduino esp32/package\_esp32\_index.json. If other board URLs are present (e.g., for ESP8266), separate them with a comma.60
* Click "OK" to save preferences.
* Go to Tools > Board > Boards Manager.
* Search for "ESP32" and click "Install" for "ESP32 by Espressif Systems".60
* After installation, select the appropriate ESP32 board (e.g., "ESP32 Dev Module") and the correct COM port under Tools > Board and Tools > Port respectively.60

3. Required Libraries Installation: Several libraries are essential for interfacing with sensors, managing Wi-Fi, and integrating with Firebase. These are typically installed via the Arduino Library Manager (Sketch > Include Library > Manage Libraries):

* **WiFi.h:** This is a built-in Arduino library for ESP32, essential for Wi-Fi connectivity.62 It provides functions for connecting to Wi-Fi networks, managing connections, and accessing network information.63
* **FirebaseESP32.h (or FirebaseClient.h):** This library facilitates communication with Google Firebase services, including the Realtime Database. It allows the ESP32 to send and receive data from Firebase.62 It supports various Firebase products like Realtime Database, Cloud Firestore, Storage, and Cloud Messaging.64
* **DHT.h:** This library is specifically for DHT series temperature and humidity sensors (like DHT11). It provides functions to read temperature and humidity values from the sensor.62
* **LiquidCrystal\_I2C.h:** This library is necessary for controlling 16x2 I2C LCD displays. It simplifies communication over the I2C protocol, allowing easy text display and backlight control.48
* **Wire.h:** This is the standard Arduino library for I2C communication, often included implicitly by LiquidCrystal\_I2C.h or required for other I2C sensors.48
* **AsyncTCP.h and ESPAsyncWebServer.h (for Web Server):** If a local web server is implemented, these libraries are crucial for creating asynchronous web servers and handling WebSocket communication on the ESP32.68
* **LittleFS.h (for Web Server):** Used to manage the ESP32's flash memory filesystem, where HTML, CSS, and JavaScript files for the web interface are stored.68
* **Arduino\_JSON.h (for Web Server):** Facilitates the creation and parsing of JSON objects for data exchange, especially useful for sending sensor readings to a web interface.68

Once these components are installed and configured, the Arduino IDE is ready for developing the ESP32 firmware for the air pollution monitoring system.

**5.2 Wi-Fi Connectivity and Network Configuration**

Wi-Fi connectivity is fundamental to an IoT system, enabling the ESP32 to transmit sensor data to cloud platforms and facilitate remote monitoring. The ESP32 can operate in two primary Wi-Fi modes: Station (STA) mode and Access Point (AP) mode. For this system, STA mode is primarily utilized to connect to an existing Wi-Fi network.

1. Connecting in Station (STA) Mode (Wi-Fi Client Mode):

In STA mode, the ESP32 acts as a client, connecting to an existing Wi-Fi network provided by an Access Point (e.g., a home router). This is the most common mode for IoT devices that need to send data to the internet.

Network Credentials: The ESP32 requires the Wi-Fi network's Service Set Identifier (SSID) and password to establish a connection. These are typically defined as constants in the code.62

C++

const char\* ssid = "YOUR\_WIFI\_SSID";

const char\* password = "YOUR\_WIFI\_PASSWORD";

Initiating Connection: The WiFi.begin() function is used to start the connection process.62

C++

WiFi.begin(ssid, password);

Connection Status Check: A loop is typically used to monitor the connection status until WL\_CONNECTED is achieved, indicating a successful connection.62

C++

while (WiFi.status()!= WL\_CONNECTED) {

delay(1000); // Wait for 1 second

Serial.print("."); // Print a dot to indicate waiting

}

Serial.println("WiFi connected");

IP Address Access: Once connected, the ESP32 is assigned an IP address by the network's DHCP server. This IP address can be retrieved and printed to the Serial Monitor using WiFi.localIP().63 This IP address is crucial for accessing any web server hosted on the ESP32 from a local browser.

C++

Serial.print("IP address: ");

Serial.println(WiFi.localIP());

For more persistent access, especially in development, it is advisable to configure the router's DHCP to always assign the same IP address to the ESP32's MAC address, or to use mDNS (Multicast DNS) to access the ESP32 by a hostname (e.g., http://esp32.local) instead of an IP address.70

Reconnection Logic: The Firebase.reconnectWiFi(true) function, part of the Firebase Arduino Client Library, ensures that the ESP32 attempts to reconnect to the Wi-Fi network if the connection is lost, enhancing system robustness.62

2. Access Point (AP) Mode (Soft-AP Mode):

While STA mode is primary, the ESP32 can also be configured as an Access Point, creating its own Wi-Fi network. This mode is useful for initial configuration, local control, or situations where no existing Wi-Fi network is available.

WiFi.softAP(ssid, password); can be used to set up the AP.63

Functions like WiFi.softAPIP() allow retrieval of the AP's IP address.63. The implementation ensures reliable wireless communication, a cornerstone for any IoT application, allowing the air quality monitoring system to send data to cloud services and be managed remotely.

### **5.3** Cloud Integration: Firebase Realtime Database

Cloud integration is a cornerstone of modern IoT systems, providing remote data storage, real-time synchronization, and accessibility from various client devices. The Firebase Realtime Database, a NoSQL cloud-hosted database, is chosen for this system due to its real-time capabilities, flexible data structure, and robust security features.

**5.3.1. Firebase Project Setup**

Setting up a Firebase project is a prerequisite for cloud integration, involving several key steps to configure the database, authentication, and access credentials.

Create a Firebase Project:

Access the Firebase Console ([console.firebase.google.com](https://console.firebase.google.com/" \t "_blank)) and sign in with a Google Account.66

Create a new project, providing a suitable name (e.g., "AirQualityMonitor").66

Google Analytics can optionally be disabled if not required for this specific project.66

Set Authentication Methods:

Authentication is critical for identifying users (including the ESP32 as a "user") and securely managing data access.66

In the Firebase Console, navigate to Build > Authentication, then select "Get started".66

Enable the "Email/Password" authentication method and save the changes.66

Under the "Users" tab, add a new user by providing an email and password. This user's credentials will be used by the ESP32 to authenticate with Firebase. Firebase automatically assigns a unique User ID (UID) to each registered user, which is crucial for database access control.66

Get Project API Key:

The project's API key is required for the ESP32 to interact with Firebase services.66

This key can be found in Project Settings within the Firebase Console. It should be copied and securely stored for use in the ESP32 code.66

Set up Realtime Database:

In the Firebase Console, navigate to Build > Realtime Database, then click "Create Database".66

Select the database location closest to the deployment region.

Initially, select "Start in test mode" for security rules, as these will be modified to implement fine-grained access control.66

After creation, copy and save the database URL, as it will be needed in the ESP32 code.66

Set up Database Security Rules:

Firebase Realtime Database Security Rules determine who has read and write access to the database, how data is structured, and what indexes exist.71 By default, access is denied to protect the database.71

To implement user-specific access, especially for IoT devices, rules are defined to allow an authenticated user to read and write data only to nodes matching their unique UID. This ensures data privacy and prevents unauthorized access to other users' or devices' data.66

Navigate to the "Rules" tab within the Realtime Database section. The following rule structure is recommended:

JSON

{

"rules": {

"UsersData": {

"$uid": {

".read": "$uid === auth.uid",

".write": "$uid === auth.uid"

}

}

}

}

This rule ensures that an authenticated user (represented by auth.uid) can only read from and write to the UsersData node under their specific $uid (User ID). Any data outside this specific node will be inaccessible to them.66 Rules are declarative and cascade top-down, meaning a rule granting access at a parent node also grants access to all child nodes, but .validate rules do not cascade.71

**5.3.2. Data Logging Logic**

The core of the cloud integration involves the ESP32's logic for acquiring sensor data, formatting it, and securely transmitting it to the Firebase Realtime Database.

ESP32 Authentication:

Upon startup, the ESP32 first attempts to connect to the configured Wi-Fi network.

Once connected, it authenticates with Firebase using the pre-configured email and password of the dedicated Firebase user.66 This authentication process is critical for gaining authorized access to the database as per the defined security rules.

After successful authentication, the ESP32 retrieves the unique User ID (UID) associated with its authenticated user. This UID is then used to construct the specific database path where the sensor data will be stored, ensuring that data is written to the correct, authorized location (e.g., UsersData/<USER\_UID>/readings/).66

Periodic Data Acquisition:

The ESP32 is programmed to periodically acquire readings from all connected sensors (MQ-series, DHT11). A typical data acquisition interval might be set to every 10 to 30 seconds, balancing real-time monitoring needs with power consumption and data volume.68

For each reading cycle, the ESP32 reads the analog values from the MQ sensors, converts them to PPM using calibration formulas, and reads digital temperature and humidity values from the DHT11.

Timestamping:

To provide a historical record and enable time-series analysis, each set of sensor readings is associated with a timestamp. The ESP32 obtains the current epoch time (Unix timestamp) from a Network Time Protocol (NTP) server.66 This ensures accurate and synchronized timestamps for all logged data.

JSON Structure for Multiple Sensor Data:

All collected sensor data (e.g., CO2 PPM, LPG PPM, Smoke PPM, Temperature, Humidity) along with the timestamp are formatted into a JSON (JavaScript Object Notation) object.62 JSON is a lightweight, human-readable data interchange format widely supported by Firebase and web applications.

Firebase Realtime Database stores data as a JSON tree. While it allows deep nesting, best practices recommend keeping data structures as flat as possible to avoid downloading entire subtrees when only a subset of data is needed.15

For sensor data, a structure like UsersData/<USER\_UID>/readings/<timestamp\_or\_unique\_id> is recommended, where each timestamped entry contains the readings from all sensors as key-value pairs.66

JSON

{

"UsersData": {

"YOUR\_USER\_UID": {

"readings": {

"1678886400": { // Epoch timestamp

"temperature": 25.5,

"humidity": 60.2,

"co2\_ppm": 750,

"lpg\_ppm": 500,

"smoke\_ppm": 100,

"timestamp": "2023-03-15T10:00:00Z"

},

"1678886430": {

//... next set of readings

}

}

}

}

}

This flattened structure under readings allows for efficient retrieval of specific time ranges or sensor types without downloading all historical data.73

Data Update to Firebase:

The formatted JSON object is then sent to the Firebase Realtime Database using the Firebase Arduino Client Library's Firebase.updateNode() or Firebase.set() functions.62

The data is pushed to the specific path associated with the authenticated user's UID, ensuring compliance with security rules.66

The Firebase.setwriteSizeLimit() function can be used to manage the size of data written in a single operation, preventing timeouts for large data transfers.62

This systematic approach to data logging ensures that the air pollution monitoring system reliably collects, timestamps, and stores critical environmental data in the cloud, making it accessible for analysis and decision-making.

**5.3.3. Firebase Arduino Client Library**

The Firebase Arduino Client Library for ESP8266 and ESP32 is a crucial software component that simplifies the interaction between the microcontroller and Google's Firebase services. This library abstracts the complexities of HTTP requests and data formatting, allowing developers to focus on application logic.

Features and Compatibility:

Comprehensive Firebase Support: The library provides functionality for interacting with various Firebase products, including:

Realtime Database: Allows for basic CRUD (Create, Read, Update, Delete) operations on the cloud-hosted JSON tree.65 This includes putting (setting), updating, getting (retrieving), and removing data at specified paths.65

Authentication: Supports user authentication methods such as signing up, signing in, resetting passwords, verifying email addresses, and deleting users.65 This is vital for securing data access based on user identity.

Cloud Firestore, Firebase Storage, Google Cloud Storage, Cloud Functions for Firebase, and Cloud Messaging: While the primary focus for this project is Realtime Database, the library's broader support offers extensibility for future enhancements.64

HTTP Requests Abstraction: It handles the underlying HTTP requests to Firebase's REST API for authentication and database operations, simplifying the communication process.65

Debugging Support: The library includes configurable debug levels, allowing developers to switch between production and debugging modes. This enables detailed logging of HTTP requests, responses, and errors, which is invaluable during development and troubleshooting.65

Wi-Fi Reconnection: The Firebase.reconnectWiFi(true) function automatically attempts to re-establish the Wi-Fi connection if it is lost, enhancing the system's robustness and continuous operation.62

Data Size Limits: Functions like Firebase.setwriteSizeLimit() allow setting limits on the data size written to the database, preventing timeouts for potentially large data transfers.62

Compatibility: The library is highly compatible with Espressif ESP8266 and ESP32 boards. It also supports other Arduino devices using standard Client interfaces (e.g., WiFiClient), making it versatile across different hardware platforms.64

Usage Examples (Simplified):

The library's usage typically involves:

Including Headers:

C++

#include <FirebaseESP32.h> // or <FirebaseClient.h> depending on version

#include <WiFi.h>

Firebase Configuration: Defining the Firebase Host URL and Authentication Key (Web API Key).62

C++

#define FIREBASE\_HOST "YOUR\_DATABASE\_URL"

#define FIREBASE\_AUTH "YOUR\_WEB\_API\_KEY"

// Initialize FirebaseData object and FirebaseJson object

FirebaseData firebaseData;

FirebaseJson json;

Authentication and Initialization in setup():

C++

void setup() {

//... Wi-Fi connection setup...

Firebase.begin(FIREBASE\_HOST, FIREBASE\_AUTH);

Firebase.reconnectWiFi(true);

// Optional: Firebase.setReadTimeout(firebaseData, 1000 \* 60);

// Optional: Firebase.setwriteSizeLimit(firebaseData, "tiny");

//... User authentication (if security rules require it)...

// Firebase.auth.signIn(firebaseData, USER\_EMAIL, USER\_PASSWORD);

}

Data Logging in loop():

C++

void loop() {

//... Read sensor data (Temperature, Humidity, PPMs)...

// Construct JSON object

json.set("/temperature", Temperature);

json.set("/humidity", Humidity);

json.set("/co2\_ppm", CO2\_PPM);

// Update node in Firebase

if (Firebase.updateNode(firebaseData, "/SensorData/YOUR\_USER\_UID/readings/" + String(epochTime), json)) {

Serial.println("Data sent to Firebase successfully.");

} else {

Serial.print("Failed to send data: ");

Serial.println(firebaseData.errorReason());

}

delay(DATA\_UPLOAD\_INTERVAL\_MS);

}

(Note: The path /SensorData/YOUR\_USER\_UID/readings/ is a common pattern for user-specific data logging, aligning with Firebase security rules.)

Known Issues: Some users have reported delayed responses or initial failures with authentication requests, which often resolve after retries. This can be mitigated by introducing delays or adjusting HTTP client connection timeouts. Stable Wi-Fi connectivity is paramount for reliable communication with Firebase services.65. The Firebase Arduino Client Library significantly streamlines the development of cloud-connected IoT applications, allowing the air pollution monitoring system to leverage Firebase's powerful real-time database capabilities with relative ease.

**5.4 Local Data Display: LCD Interface**

Providing immediate, local feedback on air quality parameters is crucial for user awareness, especially in the absence of continuous remote access. The 16x2 I2C LCD display serves this purpose, offering a simple yet effective visual interface.

Arduino Library Usage:

The LiquidCrystal\_I2C library simplifies the process of controlling the I2C LCD. This library abstracts the complexities of the I2C communication protocol, allowing developers to send commands and display text with straightforward functions.

Library Inclusion and LCD Initialization:

The library header is included at the beginning of the Arduino sketch:

C++

#include <LiquidCrystal\_I2C.h>

An LCD object is then created, specifying its I2C address (commonly 0x27 or 0x3F), the number of columns (16), and the number of rows (2).48

C++

LiquidCrystal\_I2C lcd(0x27, 16, 2); // Address, columns, rows

Initialization in setup() Function:

The setup() function, which executes once at startup, is used to initialize the LCD.

lcd.init(): Initializes the LCD module.48

lcd.clear(): Clears any existing content on the display, ensuring a clean start.48

lcd.backlight(): Turns on the LCD's backlight, making the display visible.48

Initial messages, such as a system startup message or "Air Quality Monitor," can be displayed here.48

C++

void setup() {

//... (Serial and Wi-Fi initialization)...

lcd.init();

lcd.clear();

lcd.backlight();

lcd.setCursor(0, 0); // Set cursor to column 0, row 0

lcd.print("AQ Monitoring");

lcd.setCursor(0, 1); // Set cursor to column 0, row 1

lcd.print("Initializing...");

delay(2000);

lcd.clear();

}

Displaying Real-time Data in loop() Function:

Within the loop() function, which executes repeatedly, the latest sensor readings are displayed.

lcd.setCursor(column, row): Positions the cursor at a specific column and row before printing.48

lcd.print("message") or lcd.print(variable): Prints text or numerical values to the LCD.48

To display multiple parameters, the screen can be updated cyclically or by dedicating specific areas for each reading. For example, temperature and humidity on one line, and a gas concentration on another.

C++

void loop() {

//... (Read sensor data and process)...

lcd.setCursor(0, 0);

lcd.print("Temp: " + String(temperature) + "C");

lcd.setCursor(0, 1);

lcd.print("Hum: " + String(humidity) + "%");

// Add logic to switch display or show other gas readings

delay(5000); // Display for 5 seconds

lcd.clear();

lcd.setCursor(0, 0);

lcd.print("CO2: " + String(co2\_ppm) + "ppm");

lcd.setCursor(0, 1);

lcd.print("AQI: " + String(aqi\_value)); // Assuming AQI calculation

delay(5000);

lcd.clear();

}

It is important to adjust the contrast potentiometer on the back of the I2C LCD module after wiring and power-up to ensure the displayed text is clear and visible.48

The LCD interface provides an immediate and accessible means for local monitoring,

**5.5 Summary**

The 16x2 I2C LCD display offers immediate, local air quality feedback, enhancing user awareness without relying on remote access. Using the LiquidCrystal\_I2C library, the display is easily controlled through simple commands. In the setup() function, the LCD is initialized, cleared, and backlit, often showing a startup message. In the loop() function, real-time sensor data—such as temperature, humidity, CO₂ levels, and AQI—is cyclically displayed by positioning the cursor and printing updated values. The contrast knob must be adjusted for clarity. This local display complements cloud storage, ensuring users are informed even without internet access.

**Chapter 6**

**TESTING AND RESULTS**

**6.1 Test Environment and Procedure**

**Test Environment**: The test environment for evaluating the IoT-based air pollution monitoring system was carefully selected to replicate both indoor and semi-outdoor conditions, ensuring reliable performance analysis. Testing was conducted in a closed room to monitor ambient air quality under controlled conditions and later extended to a ventilated corridor and balcony area for exposure to varied pollutant levels such as smoke and vehicular emissions. The system was powered using a 5V regulated supply via a buck converter, ensuring safe and consistent voltage to all components.Prior to testing, the MQ2, MQ6, and MQ135 gas sensors were preheated for approximately 2–3 minutes to stabilize their outputs. The DHT11 sensor provided baseline temperature and humidity levels. For simulation, test pollutants included household LPG gas (for MQ6), cigarette smoke (for MQ2), and general ambient air pollutants (for MQ135).

The LCD displayed real-time data, helping observe sensor response visually, while all data was also logged remotely on Firebase Realtime Database for cloud verification. The buzzer was tested under threshold breach conditions to confirm alert functionality.This structured test environment allowed thorough observation of the system’s performance, accuracy, responsiveness, and stability under various pollutant exposures and environmental changes, confirming its suitability for practical air quality monitoring applications.

**Test Procedure**: The validation process began with assembling the system and calibrating all sensors in a clean indoor environment. The setup was then checked for accurate readings and proper functionality of the LCD, buzzer, and Firebase data logging. Smoke was introduced to test the responsiveness of the MQ2 and MQ135 sensors and the triggering of alerts. A gas leak was simulated to evaluate the MQ6 sensor’s accuracy and system behavior. The device was then placed outdoors to observe its performance in natural atmospheric conditions. Further testing was done near traffic to assess the sensors under high pollution levels. Finally, the system was run continuously for 24 hours to ensure stability, power efficiency, and reliable data logging.

**6.2 Functional Testing (Scanning Accuracy)**

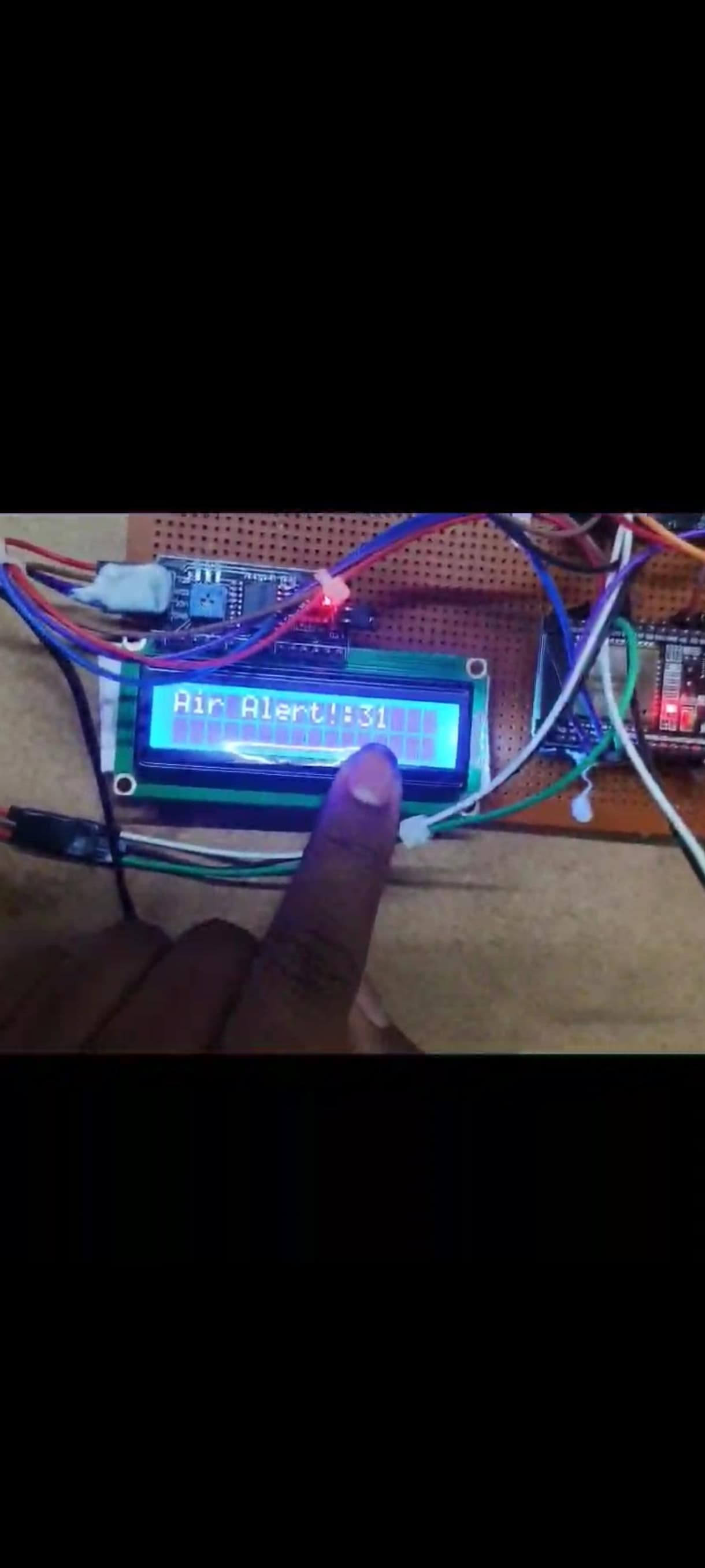
· **Sensor Functionality Verification**:  
All environmental sensors—MQ135, MQ2, MQ6, and DHT11—were tested individually to confirm accurate data acquisition. The system responded correctly to different gas exposures (like smoke and LPG), and the DHT11 reliably provided temperature and humidity values under varied conditions.

· **System Output and Alert Mechanism**:  
The 16x2 I2C LCD displayed real-time data effectively, and the buzzer activated whenever gas concentration levels exceeded safe thresholds. These outputs ensured users received immediate and clear alerts both visually and audibly.

· **Cloud Connectivity and Data Handling**:  
The ESP32's Wi-Fi functionality was tested for stable connection to the Firebase Realtime Database. Sensor data was successfully uploaded, stored, and could be monitored remotely, confirming end-to-end functionality from sensing to cloud reporting.

**6.3 Sample Output Snapshots**

The project succesfully demonstrated then output,when any gas level exceeded the present threshold (e.g., 40%), the onboard buzzer was activated and an alert flag was updated on Firebase. The LCD dynamically updated its display based on the gas readings and alert condition.



**Fig. 6.1: Simulation result**



**Fig. 6.2: Simulation result**

The image displays a functioning IoT-based air pollution monitoring system built on a perforated PCB. The setup features an ESP32 microcontroller, a 16x2 I2C LCD display, gas sensors (such as MQ2, MQ6, and MQ135), and a DHT11 sensor for temperature and humidity. The LCD shows real-time readings: Gas at 30%, Temperature at 31°C, LPG at 4%, and CO2 at 37%. These values are processed by the ESP32 and shown locally for quick access. The red LED and organized wiring indicate proper operation and assembly. This compact system is ideal for indoor air quality monitoring and environmental data logging applications.

**6.5 Summary**

The testing and results phase of the IoT-based air pollution monitoring system focused on evaluating sensor accuracy, system reliability, data transmission, and user alerts. Functional testing confirmed that the MQ135, MQ2, and MQ6 sensors successfully detected various harmful gases like smoke, CO₂, and LPG. The DHT11 sensor provided accurate temperature and humidity readings, essential for correlating environmental conditions with air quality.The system consistently displayed real-time data on the 16x2 I2C LCD and triggered the buzzer when pollutant levels exceeded safe limits. Data was successfully uploaded to Firebase Realtime Database, enabling remote access and long-term monitoring. Different environments—indoors, near traffic, and during controlled exposure to smoke and gas—were used to test performance under real-world conditions.Stability tests showed that the ESP32-based system maintained continuous operation without data loss or power-related issues, thanks to the buck converter. Overall, the system proved to be reliable, responsive, and suitable for continuous air quality monitoring, offering immediate local feedback and cloud-based analytics. These results validate the project's effectiveness in detecting and reporting environmental conditions for improved public awareness and safety.

**Chapter 7**

**CONCLUSION AND FUTURE SCOPE**

**7.1 Conclusion**

This project successfully demonstrated the design and implementation of an IoT-based air pollution monitoring system using ESP32, integrated gas sensors (MQ135, MQ2, MQ6), DHT11 for temperature and humidity, and Firebase for cloud storage. The system effectively monitors harmful gases and environmental parameters in real-time, provides immediate alerts through a buzzer and LCD, and ensures remote access to data through cloud connectivity. It is compact, cost-effective, and well-suited for practical deployment in both indoor and outdoor environments.

Comprehensive Monitoring: The integration of multiple gas sensors and a temperature-humidity sensor ensures a broad and accurate measurement of air quality parameters.

Effective Real-Time Alerts: Local output devices like a buzzer and LCD provide instant alerts when pollution crosses predefined safe thresholds.

User-Friendly Design: The use of Arduino IDE and prebuilt libraries allowed for easy programming, LCD interfacing, and modular system updates.

Low-Cost and Educational Value: Using readily available components makes the system ideal for student projects, academic research, and community applications.

**7.2 Limitations**

Despite the successful implementation and functionality of the IoT-based air pollution monitoring system, certain limitations were identified during development and testing. These limitations impact the system's performance, accuracy, and scalability in real-world applications.

**Sensor Calibration Needs:**  
The MQ-series sensors are sensitive but require frequent calibration for accurate gas concentration readings. Lack of calibration can result in fluctuating or incorrect data.

**Limited Gas Detection Range:**Each sensor (e.g., MQ135, MQ2, MQ6) is tuned to specific gases. The system cannot detect pollutants outside the target gases without additional sensor integration.

**No Battery Backup:**The system relies on a stable external power supply. Without a battery or UPS, it cannot function during power outages.

**No GPS or Location Tagging:**The system does not provide geo-tagged data, which limits its usefulness in large-scale deployment across multiple locations.

**Wi-Fi Dependency:**Continuous Wi-Fi is required for Firebase connectivity. In areas with weak or no internet, data transmission is interrupted.

**Short-Term Durability:**Sensors like MQ135 can degrade over time due to continuous exposure, affecting their longevity and accuracy.

**7.3 Future Scope**

**Integration with Mobile Applications**:  
A custom Android or iOS mobile app can be developed to receive real-time alerts, view historical data, and monitor live air quality remotely. This would enhance user interaction and accessibility, making the system more user-friendly for non-technical individuals.

**Deployment of GPS Module**:  
Adding a GPS module to the ESP32 would enable geo-tagging of air quality data. This is especially useful for mobile monitoring units like those installed in vehicles or drones, allowing authorities to create pollution heatmaps for specific regions.

**Solar-Powered Operation**:  
To make the system self-sustainable and suitable for outdoor or remote deployment, a solar power setup with battery backup can be implemented. This eliminates the need for continuous external power and enhances environmental adaptability.

**Advanced Sensor Array and AI-Based Analysis**:  
Future versions can integrate more precise sensors (like PM2.5, NOx, CO2) and use AI or machine learning algorithms to predict pollution patterns, identify pollutant sources, and suggest mitigation strategies. This makes the system proactive rather than just reactive.

**7.4 Summary**

This final chapter provides a comprehensive summary of the IoT-based air pollution monitoring system, highlighting its design, development, and implementation. The project successfully demonstrates the integration of smart sensors with an ESP32 microcontroller, Firebase cloud storage, and an LCD interface for real-time monitoring of environmental conditions. By employing gas sensors such as MQ135, MQ2, and MQ6 along with the DHT11 sensor, the system accurately detects harmful gases, temperature, and humidity levels. Alerts are issued locally via a buzzer and globally through the cloud, ensuring user awareness and data accessibility. A buck converter ensures regulated power supply, maintaining system stability. The modular architecture allows easy upgrades and adaptability to different environments. Testing validated the system’s core functionality, responsiveness, and accuracy. Although certain limitations exist, such as limited sensing range and Wi-Fi dependency, the project lays a strong foundation for future enhancements involving GPS, mobile apps, and AI analytics for broader applications.

**REFERENCES**

[1] N. Sharma and A. Singh, “Design and Development of IoT-Based Air Pollution Monitoring System,” M.Tech Thesis, Department of Electronics and Communication, National Institute of Technology, 2020.

[2] R. Kumar, S. Verma, and T. Jain, “Real-Time Air Quality Monitoring Using IoT and Firebase,” B.Tech Project Report, Department of Computer Engineering, VIT University, 2021.

[3] A. Mehta and D. Reddy, “IoT-Based Environmental Monitoring with Cloud Integration,” Undergraduate Thesis, SRM Institute of Science and Technology, 2022.

[4] P. Lakshmi and R. Karthik, “Design and Implementation of a Low-Cost Air Quality Monitoring System Using ESP32,” B.E. Thesis, Anna University, 2019.

[5] S. G. Patil and A. Deshmukh, “Wireless Sensor Networks for Urban Air Quality Monitoring Using MQTT and Firebase,” M.Sc. Thesis, University of Pune, 2020.

[6] P. S. Desai and K. S. Patel, “Air Quality Monitoring System Using IoT,” 2021 7th International Conference on Advanced Computing and Communication Systems (ICACCS), Coimbatore, India, 2021, pp. 1885–1889. DOI: 10.1109/ICACCS51430.2021.9441866.

**APPENDIX: FULL ANNOTATED SOURCE CODE**

#include <WiFi.h>

#include <FirebaseESP32.h>

#include <Wire.h>

#include <LiquidCrystal\_I2C.h>

#include <DHT.h>

// --- WiFi Credentials ---

#define WIFI\_SSID "air"

#define WIFI\_PASSWORD "123456789"

// --- Firebase Credentials ---

#define API\_KEY "AIzaSyAB5ErkFfIJgjlp8-ZqiKW7-OFlkFMT1Aw"

#define DATABASE\_URL "https://project-nsic-default-rtdb.asia-southeast1.firebasedatabase.app/"

#define USER\_EMAIL "example@gmail.com"

#define USER\_PASSWORD "123456789"

// --- Sensor Pins ---

#define MQ2\_PIN 34

#define MQ6\_PIN 35

#define MQ135\_PIN 32

#define DHT\_PIN 4

#define BUZZER\_PIN 15

// --- DHT Setup ---

#define DHTTYPE DHT11

DHT dht(DHT\_PIN, DHT11);

// --- LCD Setup ---

LiquidCrystal\_I2C lcd(0x27, 16, 2);

// --- Firebase Setup ---

FirebaseData fbdo;

FirebaseAuth auth;

FirebaseConfig config;

void connectToWiFi() {

lcd.clear();

lcd.setCursor(0, 0);

lcd.print("Connecting...");

WiFi.begin(WIFI\_SSID, WIFI\_PASSWORD);

while (WiFi.status() != WL\_CONNECTED) {

delay(500);

Serial.print(".");

}

lcd.clear();

lcd.setCursor(0, 0);

lcd.print("WiFi Connected");

lcd.setCursor(0, 1);

lcd.print(WiFi.localIP());

delay(2000);

}

void setFirebase() {

auth.user.email = USER\_EMAIL;

auth.user.password = USER\_PASSWORD;

config.api\_key = API\_KEY;

config.database\_url = DATABASE\_URL;

Firebase.begin(&config, &auth);

}

void setup() {

Serial.begin(115200);

lcd.begin();

lcd.backlight();

dht.begin();

pinMode(BUZZER\_PIN, OUTPUT);

connectToWiFi();

setFirebase();

}

void loop() {

if (Firebase.ready()) {

// Raw analog values

int raw\_mq2 = analogRead(MQ2\_PIN);

int raw\_mq6 = analogRead(MQ6\_PIN);

int raw\_mq135 = analogRead(MQ135\_PIN);

// Convert to percentages

int mq2 = map(raw\_mq2, 0, 4095, 0, 100);

int mq6 = map(raw\_mq6, 0, 4095, 0, 100);

int mq135 = map(raw\_mq135, 0, 4095, 0, 100);

float temperature = dht.readTemperature();

float humidity = dht.readHumidity();

// Display on Serial

Serial.printf("MQ2: %d%%, MQ6: %d%%, MQ135: %d%%, Temp: %.1fC, Humidity: %.1f%%\n",

mq2, mq6, mq135, temperature, humidity);

// Upload to Firebase

Firebase.setInt(fbdo, "/air/MQ2", mq2);

Firebase.setInt(fbdo, "/air/MQ6", mq6);

Firebase.setInt(fbdo, "/air/MQ135", mq135);

Firebase.setFloat(fbdo, "/air/Temperature", temperature);

Firebase.setFloat(fbdo, "/air/Humidity", humidity);

// Alert logic

bool alert = mq2 > 40 || mq6 > 40 || mq135 > 40;

digitalWrite(BUZZER\_PIN, alert ? HIGH : LOW);

Firebase.setInt(fbdo, "/air/Alert", alert ? 1 : 0);

// Display on LCD

lcd.clear();

lcd.setCursor(9, 0);

lcd.print("T:");

lcd.print(temperature, 0);

//lcd.setCursor(8, 0);

//lcd.print("H:");

//lcd.print(humidity, 0);

lcd.setCursor(0, 1);

if (alert) {

lcd.setCursor(0, 0);

lcd.print("Air Alert!");

} else {

lcd.print("LPG:");

lcd.print(mq2);

lcd.print("%");

}

lcd.setCursor(0, 0);

if (alert) {

lcd.setCursor(0, 0);

lcd.print("Air Alert!");

} else {

lcd.print("gas:");

lcd.print(mq6);

lcd.print("%");

}

lcd.setCursor(8, 1);

if (alert) {

lcd.setCursor(0, 0);

lcd.print("Air Alert!");

} else {

lcd.print("CO2:");

lcd.print(mq135);

lcd.print("%");

}

delay(5000); // Delay before next read

}

}