Ahmedabad University

Project Report-1
Group-8
Deadline: 12/09/2024

Instructor: Anurag Lakhlani **Teaching Assistant:** Priyesh Chaturvedi

Enrolment No	Name
AU2240118	Tirth Teraiya
AU2240080	Namra Maniar
AU2240244	Saumya Shah
AU224075	Henish Trada

Chapter 01

- Introduction

Air pollution has become one of the most pressing environmental and health issues of the 21st century, mainly due to industrial emissions, vehicular exhaust, and urbanization. While eliminating air pollution entirely may seem challenging, reducing its harmful effects through continuous monitoring and preventive measures is achievable. Many densely populated cities, particularly in developing nations, face severe air quality deterioration without adequate monitoring systems in place. This leads to respiratory illnesses, cardiovascular problems, and long-term health risks for the population. To address this growing concern, a system is being developed that simplifies air quality monitoring and classification, focusing on essential parameters such as particulate matter (PM2.5 and PM10), carbon monoxide (CO), nitrogen dioxide (NO_2), ozone (O_3), and temperature/humidity levels.

The advancement of the Internet of Things (IoT) has introduced new possibilities for real-time environmental data collection, analysis, and remote monitoring. This technology enables users to track air quality trends in real-time from any location. The proposed system will consist of multiple nodes, each equipped with an array of sensors, deployed across urban and industrial regions. These sensors will continuously collect data on air pollutants and wirelessly transmit it to a central server via IoT communication protocols. If pollutant levels exceed safe thresholds, an automated alert will be triggered to notify concerned authorities and residents. The collected data will also be analyzed using classification algorithms to categorize air quality into health-based categories, aiding decision-makers in implementing mitigation strategies.

Different sensors play a critical role in monitoring various aspects of air quality. For instance, a dust sensor measures PM2.5 and PM10 concentrations, a gas sensor detects CO and NO_2, while ozone sensors track O_3 levels. A DHT11 or DHT22 sensor measures temperature and humidity, as these factors influence pollutant dispersion. An Arduino or ESP32 microcontroller collects the sensor data, and with the help of an ESP8266/LoRa communication module, the information is transmitted to a cloud platform for visualization. The system classifies air quality levels—such as “Good,” “Moderate,” or “Hazardous”—based on Air Quality Index (AQI) standards, making the data easy to interpret for both researchers and the general public.

Air pollution is particularly concerning in countries like India and China, where rapid industrialization and high vehicle density worsen the situation. Managing air quality and ensuring a safe environment have become urgent priorities. Traditional air monitoring methods are limited to large, expensive stations that cover only select locations, leaving gaps in real-time, widespread monitoring. This limitation often results in delayed detection of hazardous pollution

spikes, increasing health risks. IoT-based air quality monitoring offers a more scalable and cost-effective solution.

The proposed system aims to provide real-time pollutant data, generate historical trend analyses, and issue early warnings for hazardous conditions. By leveraging IoT and advanced classification techniques, the system not only detects pollutant levels but also categorizes air quality into actionable classes aligned with global AQI standards. This helps authorities take proactive measures, such as traffic control, public advisories, or industrial emission management, ultimately contributing to healthier living conditions.

Problem Definition

Air is a vital natural resource, yet its quality is rapidly deteriorating due to industrialization, urbanization, and vehicular emissions. Unfortunately, air pollution has become one of the most critical global concerns in recent decades, with hazardous gases and fine particulate matter contaminating the atmosphere. While completely eradicating air pollution is impossible, it is essential to minimize its harmful impacts. In many developing nations, people are exposed daily to unsafe air quality without proper monitoring or warnings, which leads to severe health risks. Air pollution occurs when harmful gases, chemical particles, or biological matter contaminate the atmosphere, reducing its purity and threatening the well-being of living organisms. Polluted air contributes to the death of plants and animals, disrupts ecosystems, and accelerates climate change. Moreover, exposure to polluted air causes respiratory diseases such as asthma, bronchitis, and lung cancer, leading to approximately 7 million premature deaths globally every year, according to the World Health Organization (WHO).

Motivation

The suggested system enhances air quality monitoring by introducing more advanced and efficient approaches. It emphasizes increasing the frequency of data collection, integrating more accurate and durable sensors, and expanding the range of air pollutants being measured. This project aims to create a streamlined process for gathering atmospheric data, transmitting it wirelessly, and storing it securely. A key feature is the ability to archive all collected data, which can serve as a valuable resource for future research and analysis. This will enable better tracking of pollution patterns and provide deeper insights into how air quality fluctuates across different times and locations.

Chapter 02

Market Survey or Literature Survey of Current Products

Market Size:

The global air quality monitoring market is expanding rapidly due to increasing concerns about urban air pollution, rising health issues caused by poor air quality, and stricter government regulations. According to multiple market research reports, the industry is expected to grow at a significant CAGR in the coming years. The rising adoption of IoT-enabled devices, combined with the demand for real-time monitoring, is driving the growth of this market. Additionally, the need for low-cost, portable solutions is creating opportunities for innovative IoT-based air monitoring systems.

Relevant Products:

Given the precise needs specified in our proposal, a variety of items apply to the market:

1. IoT-based Air Quality Monitoring Systems

Features: These systems integrate sensors to measure air quality parameters such as PM2.5, PM10, CO_2, CO, SO_2, NO_2, O_3, temperature, and humidity. They utilize IoT technology to transmit real-time data to a cloud platform for monitoring, visualization, and analysis.
Examples: Companies like Aeroqual, TSI Incorporated, and Kaiterra provide IoT-based air quality monitoring systems for urban, industrial, and residential use.

2. Wireless Air Quality Sensors

Features: Compact and power-efficient, these sensors can be deployed in multiple locations, even in remote or high-density urban areas. They wirelessly send pollution data, making them suitable for long-term and large-scale monitoring.
Examples: Companies such as AirVisual (IQAir), Spec Sensors, and Sensirion offer wireless air quality sensors with capabilities for detecting particulate matter and harmful gases.

3. Air Quality Monitoring Software Platforms

Features: Software platforms collect, analyze, and visualize air pollution data. They support AQI classification, trend forecasting, and regulatory compliance checks. They also provide dashboards and mobile applications for public awareness and decision-making.
Examples: BreezoMeter, Airly, and Clarity Movement Co. provide software solutions tailored for real-time air quality tracking.

Market Trends:

- **Growing IoT adoption:** Integration of IoT and AI technologies is revolutionizing air quality monitoring by enabling real-time pollutant detection and AQI classification.

- **Focus on portable and remote monitoring:** Compact wireless sensors and cloud-based platforms are increasingly popular for urban, residential, and industrial monitoring.

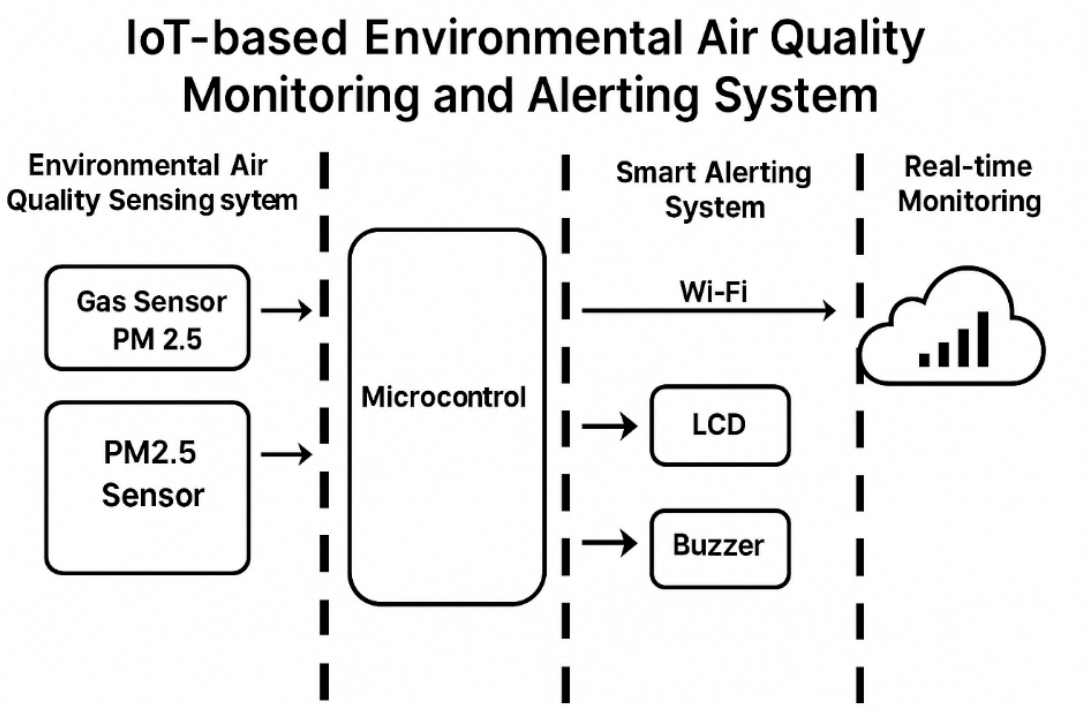
- **Advances in sensor technology:** Development of more accurate, low-cost, and energy-efficient sensors is expanding the possibilities for continuous air quality tracking.

- **Public health awareness and regulation:** Governments and organizations are pushing stricter emission norms, while the public is demanding transparent, data-driven air quality insights.

- **AI-based classification:** Advanced algorithms are being used to classify pollution levels into health-based categories, allowing better decision-making and early warnings.

Chapter 03

Block Diagram and Explanation



1. Environmental Air Quality Sensing System

This subsystem is responsible for **detecting and measuring pollutants** in the air:

- **Gas Sensor MQ135**
    - Detects harmful gases such as **CO_2, NH_3, benzene, NOx, and smoke**.
    - Provides an analog voltage signal proportional to the gas concentration.
- **Particulate Matter Sensor (PM2.5 / PM10)**
    - Measures the concentration of fine dust particles suspended in the air.

- PM2.5 = Particles ≤ 2.5 micrometers, highly dangerous as they penetrate deep into lungs.
- PM10 = Particles ≤ 10 micrometers, also harmful to respiratory health.
- Provides digital/analog output depending on sensor model (e.g., SDS011, PMS5003).

2. Processing and Smart Alerting System

- **Raspberry Pi 3 Model B**
  - Acts as the main processing unit.
  - Collects readings from MQ135, PM sensor, and DHT sensor.
  - Compares data with standard Air Quality Index (AQI) thresholds.
  - Performs classification of air quality levels (e.g., Good, Moderate, Poor, Hazardous).
  - Connects to Wi-Fi for uploading data to the cloud.
- **LCD Display**
  - Shows real-time readings of gas concentration, PM2.5/PM10 levels, temperature, and humidity.
  - Can also display the classified AQI category.
- **Buzzer**
  - Gives an alert/alarm if air quality crosses hazardous levels.
  - Acts as a local safety warning system.

3. Real-time Monitoring System

- **Wi-Fi Communication**
    - Enables Raspberry Pi to upload sensor data to the cloud or real-time database.
- **Real-time Database / Cloud Storage**
    - Stores continuous data on air quality for long-term monitoring and analysis.
    - Can be visualized later for trend analysis or reports.

Inputs to the System

- Air quality parameters from the environment:
    - Gas concentrations (CO_2, NH_3, NOx, smoke, etc.) → from MQ135
    - Particulate matter levels (PM2.5 / PM10) → from PM sensor
    - Temperature and Humidity → from DHT11/DHT22 sensor

These are the raw environmental inputs captured by the sensing system.

Outputs of the System

- Real-time Display of air quality parameters and their AQI classification (e.g., Good, Moderate, Poor, Hazardous) on LCD.
- Alerts via Buzzer if pollutants exceed safe thresholds.
- Data Upload to Cloud/Database for storage, analysis, and visualization of long-term trends.

Chapter 04

Circuit Diagram and Explanation

Introduction

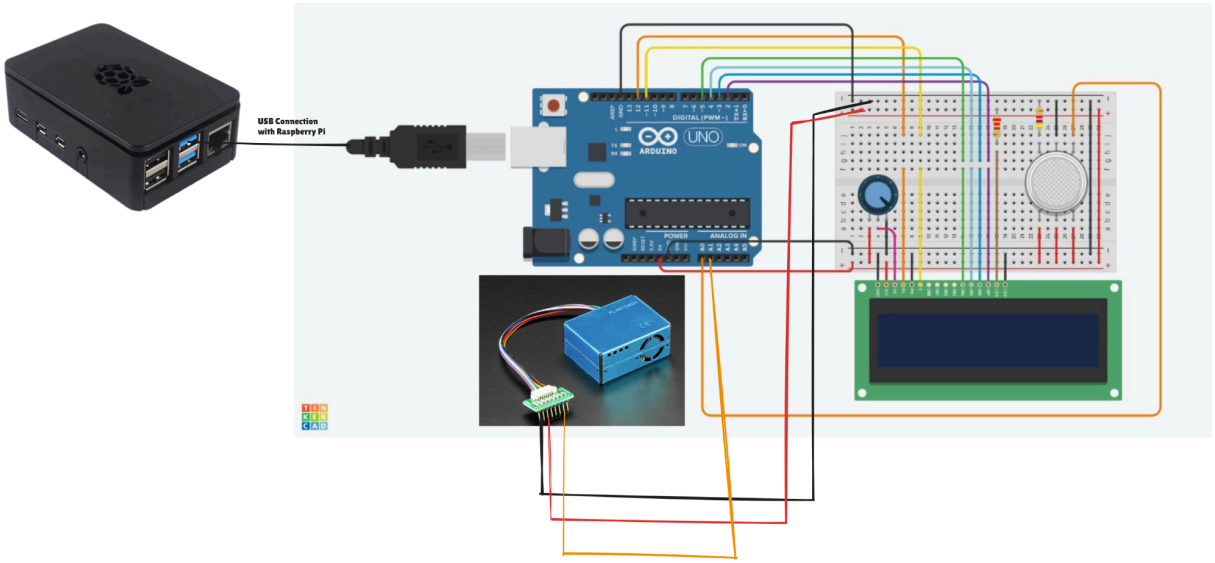
This chapter presents the detailed electronic circuit design for the IoT-based Environmental Air Quality Monitoring and Alerting System. It outlines the overall system schematic, provides a comprehensive justification for the selection of each hardware component, and analyzes the technical inputs and outputs of the integrated system.

4.1 Overall System Schematic

The system is architected around a Raspberry Pi 3 Model B, which serves as the central processing and communication hub. A critical design consideration is the Raspberry Pi's lack of native analog-to-digital conversion capabilities, which necessitates the inclusion of an external Analog-to-Digital Converter (ADC) Arduino UNO to interface with the MQ135 gas sensor and PM2.5 particle sensor. We will display the reading on LCD display

The complete wiring diagram is detailed below, describing the connections between the Raspberry Pi and all peripheral components.

Circuit Diagram :



Component Connections:

1. **Power Distribution:**
    - The Raspberry Pi's 5V pins supply power to the MCP3008 ADC, MQ135 Gas Sensor, PMS5003 PM Sensor, 16x2 I2C LCD, and the Active Buzzer module.
    - All components share a common ground (GND) connected to the Raspberry Pi's GND pins.
2. **Arduino UNO to Raspberry Pi (USB Serial Interface):** The Arduino UNO is essential for converting the analog signal from the MQ135 sensor into a digital format and sending it to the Raspberry Pi.
    - Arduino UNO's USB port is connected to one of the Raspberry Pi's USB ports.
3. **MQ135 Gas Sensor to MCP3008:**
    - MQ135 VCC to Raspberry Pi 5V.
    - MQ135 GND to Raspberry Pi GND.
    - MQ135 AOUT (Analog Output) to MCP3008 CH0 (Channel 0).
4. **PMS5003 Particulate Matter Sensor to Raspberry Pi (UART Interface):**
    - PMS5003 VCC to Raspberry Pi 5V.
    - PMS5003 GND to Raspberry Pi GND.
    - PMS5003 TXD (Transmit) to Raspberry Pi RXD (GPIO15).
    - The sensor's logic level is 3.3V, making it directly compatible with the Raspberry Pi's GPIO pins.
5. **16x2 I2C LCD to Raspberry Pi (I2C Interface):**
    - LCD VCC to Raspberry Pi 5V.
    - LCD GND to Raspberry Pi GND.
    - LCD SDA (Serial Data) to Raspberry Pi SDA (GPIO2).
    - LCD SCL (Serial Clock) to Raspberry Pi SCL (GPIO3).
6. **Active Buzzer Module to Raspberry Pi (Digital GPIO):**
    - Buzzer VCC to Raspberry Pi 5V.

- Buzzer GND to Raspberry Pi GND.
- Buzzer I/O (Signal) to Raspberry Pi GPIO17.

4.2 Component Selection and Justification

The selection of each component was a deliberate process, balancing performance requirements, cost, interfacing complexity, and alignment with the project's objectives as defined in Project Report 1. The rationale behind each choice is detailed in Table 4.1.

Component	Key Specifications	Selection Rationale for this Project
Raspberry Pi 3 B	1.2GHz Quad-Core CPU, 1GB RAM, Wi-Fi/BLE	The project requires processing data from multiple sensors concurrently and transmitting it to a cloud platform. The Raspberry Pi's powerful processor, ample RAM, and built-in Wi-Fi make it an ideal choice for this application.
Arduino UNO	ATmega328P, 6 Analog Inputs, USB Interface	The Raspberry Pi lacks native analog inputs, making it incompatible with the MQ135's analog output. An Arduino UNO is used as a cost-effective and programmable ADC. It reads the analog voltage from the sensor, converts it to a digital value, and transmits it to the Raspberry Pi via a simple USB serial connection, leveraging its built-in ADC and straightforward serial libraries.
MQ135 Sensor	Detects NH3, NOx, alcohol, Benzene, smoke, CO2 Detection Range : 100-1000ppm	This sensor directly addresses the project's primary goal of monitoring a wide range of harmful gases commonly associated with poor air quality.
PMS5003 Sensor	Measures PM1.0, PM2.5, PM10 via laser scattering	Particulate matter is a critical indicator of air pollution and a key parameter mentioned in the project's introduction. The PMS5003 was chosen for its accuracy, reliability, and its simple UART interface, which can be easily managed by the Raspberry Pi's serial port.

16x2 I2C LCD	2-line, 16-char display	This component fulfills the local display requirement of the "Processing and Smart Alerting System". The I2C version was specifically chosen to minimize the number of GPIO pins required for operation (only two data pins), preserving the Raspberry Pi's limited GPIO resources for other sensors and actuators.
Active Buzzer	Built-in oscillator, ~2.5kHz tone, low-level trigger	The buzzer provides the audible alert functionality specified in the block diagram. An active buzzer was selected for its simplicity; it generates a tone with a simple DC signal from a GPIO pin.

Table 4.1 : Component Selection Rationale

4.3 Technical Input/Output Analysis

4.3.1 System Inputs

- **Environmental Data: These are the raw physical parameters being measured from the ambient atmosphere.**
    - Concentration of gaseous compounds (e.g., ammonia, nitrogen oxides, benzene, smoke, carbon dioxide).
    - Density of suspended particulate matter, specifically PM1.0, PM2.5, and PM10.
    - Ambient air temperature.
    - Relative humidity.
- **Electrical Signals: These are the transduced signals that the Raspberry Pi processes.**
    - **Serial Data Stream (USB/UART from Arduino):** The MQ135 sensor outputs a variable analog voltage to the Arduino UNO. The Arduino converts this to a 10-bit digital value and transmits it as a serial data stream over USB to the Raspberry Pi.
    - **Serial Data Stream (UART from PMS5003):** The PMS5003 sensor transmits a structured 32-byte data frame via its TXD pin at a baud rate of 9600 bps. This frame contains discrete values for PM1.0, PM2.5, and PM10 concentrations.

4.3.2 System Outputs

The system generates three primary types of outputs to fulfill its monitoring and alerting functions.

- **Visual Information:** Real-time data is presented to the user on the 16x2 character LCD. This includes:
    - Numerical values for PM2.5 (in µg/m³), temperature (in °C), and humidity (in %).
    - A qualitative classification of the current air quality based on a calculated Air Quality Index (AQI), such as "Good," "Moderate," or "Hazardous".
- **Audible Alerts:** An audible alarm is generated by the active buzzer module.
    - This output consists of a continuous, single-frequency tone of approximately 2.5 kHz.
    - The alert is triggered programmatically when the calculated AQI surpasses a predefined "Hazardous" threshold, providing an immediate local warning of unsafe conditions.

Chapter 05

Arduino UNO/Mega OR Raspberry Pi Features

This chapter presents the detailed electronic circuit design for the IoT-based Environmental Air Quality Monitoring and Alerting System. It outlines the overall system schematic, provides a comprehensive justification for the selection of each hardware component, and analyzes the technical inputs and outputs of the integrated system.

5.1 Raspberry Pi 3 Model B: Core Features and Project Role

The Raspberry Pi 3 Model B is a credit card-sized single-board computer (SBC) that offers a powerful combination of processing capability, memory, and integrated connectivity, making it a popular choice for complex IoT projects.

5.1.1 Key Features Utilized in Project

Processor: The board is equipped with a Quad-Core 1.2GHz Broadcom BCM2837 64-bit ARMv8 processor.

Memory: It includes 1GB of LPDDR2 SDRAM.

Connectivity: The on-board BCM43438 wireless LAN (Wi-Fi 802.11n) and Bluetooth Low Energy (BLE) are cornerstone features for this IoT project.

GPIO (General Purpose Input/Output): The 40-pin extended GPIO header provides the physical interface to all sensors and actuators. This project utilizes the header's support for multiple protocols:

- **I2C pins (SDA, SCL):** For the 16x2 LCD module.
- **UART pins (TXD, RXD):** For the PMS5003 sensor.
- **Standard Digital I/O pins:** For Active Buzzer.

USB Ports: The four USB 2.0 ports are used for interfacing with the Arduino UNO, which acts as an ADC and provides data over a serial connection.

5.1.2 Strengths and Weaknesses for this Project

Strengths:

- **Full-fledged Operating System:** Running Raspberry Pi OS (a Debian derivative) greatly simplifies development. It allows for easy installation of programming languages (Python), libraries (e.g., RPi.GPIO, smbus), remote access via SSH for debugging, and robust management of network connections.
- **Vast Community and Library Support:** The Raspberry Pi has one of the largest and most active communities in the maker space. This translates to extensive, well-maintained Python libraries for virtually every common sensor and peripheral, including every component used in this project.
- **High Computing Power:** The quad-core processor enables not only data acquisition but also on-device data processing. This allows for the implementation of more complex AQI calculation algorithms.

Weaknesses and Mitigation:

- **Lack of Analog Inputs:** The most significant weakness of the Raspberry Pi for electronics projects is its inability to read analog signals directly. This limitation is directly addressed by using an Arduino UNO as an external ADC. The Arduino reads the analog signal from the MQ135, and the Raspberry Pi reads the digitized value from the Arduino over a standard USB serial connection. This is a common and well-documented pattern for interfacing analog sensors with a Raspberry Pi.
- **Higher Power Consumption:** Compared to a microcontroller like the Arduino, the Raspberry Pi is significantly more power-hungry, consuming around 700 mA or more under load. This makes it less suitable for battery-powered, portable applications. For

this project, a stationary monitoring node powered by a stable 5V/2.5A DC adapter is assumed, rendering this weakness irrelevant to the primary use case.

Chapter 06

Program Flow Chart

This chapter outlines the logical flow of the software designed to run on the Raspberry Pi. The program is responsible for initializing hardware, continuously reading data from all sensors, processing this data to determine air quality, displaying the results locally, and triggering alerts when necessary.

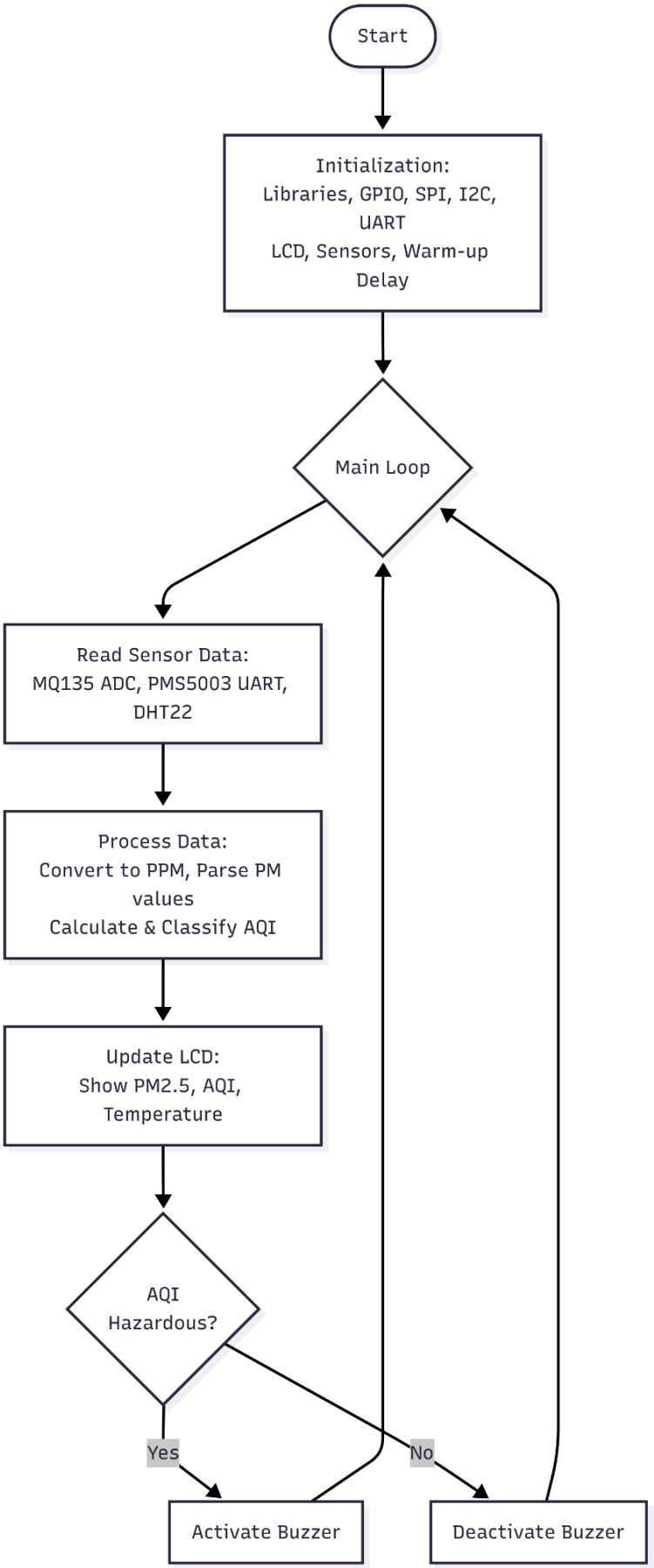
6.1 High-Level Program Logic Flowchart

The program operates on a continuous loop, orchestrating the various hardware components. A significant aspect of the software architecture is its ability to manage multiple, disparate communication interfaces concurrently—a task well-suited to the Raspberry Pi's multi-threaded Linux environment. The logical sequence of operations is as follows:

1. **Start:** The program execution begins.
2. **Initialization:** This is a critical one-time setup phase.
    - **Import Libraries:** Load necessary Python libraries.
    - **Initialize Interfaces:** Configure the GPIO pins for their respective modes (input/output). This includes setting up the I2C bus for the LCD and the serial (UART) port for the PMS5003. A separate serial object is created for the USB connection to the Arduino.
    - **Initialize Peripherals:** Create objects for each hardware component (e.g., the LCD object, the ADC object).
    - **Display Startup Message:** Write an initial message like "System Initializing..." to the LCD to provide user feedback.
    - **Sensor Warm-up:** Implement a mandatory delay to allow sensors, particularly the MQ135 with its internal heater, to reach a stable operating temperature. This is crucial for accurate initial readings.
3. **Enter Main Loop:** The program enters an infinite `while` loop to perform continuous monitoring.
4. **Read All Sensor Data:** In each iteration of the loop, the program polls each sensor to gather fresh data.
    - Read the serial data line from the Arduino UNO, which contains the 10-bit digitized value from the MQ135 sensor.
    - Read the 32-byte data frame from the PMS5003 via the UART serial buffer and validate the checksum.
5. **Process and Analyze Data:** The raw data from the sensors is converted into meaningful environmental metrics.

- Convert the raw ADC value received from the Arduino into a voltage and then, using the sensor's characteristic curve from its datasheet, estimate the gas concentration in Parts Per Million (PPM).
- Parse the PMS5003 data frame to extract the PM2.5 and PM10 concentration values (in μg/m³).
- Calculate a composite Air Quality Index (AQI) using a weighted algorithm based on the measured PM2.5 and gas concentration values.
- Classify the calculated AQI into a human-readable category (e.g., "Good," "Moderate," "Hazardous") based on standard thresholds.

6. **Update Local Display:** The processed information is displayed on the 16x2 I2C LCD.
    - The LCD is cleared.
    - Formatted strings showing the most critical data (e.g., PM2.5 value, AQI category, temperature) are written to the two lines of the display.
7. **Check Alert Condition:** A decision point checks if the air quality has reached a critical level.
    - The program evaluates: `if AQI_category == "Hazardous"`.
8. **Manage Alert System:**
    - **If True (Hazardous):** The program sends a signal to the GPIO pin connected to the active buzzer, causing it to emit an audible alarm.
    - **If False (Not Hazardous):** The program ensures the buzzer's GPIO pin is in a state that keeps the buzzer silent.
9. **Loop:** The execution flow returns to the "Read All Sensor Data" step to begin the next monitoring cycle.


Start
Initialization:
Libraries, GPIO, SPI, I2C,
UART
LCD, Sensors, Warm-up
Delay
Main Loop
Read Sensor Data:
MQ135 ADC, PMS5003 UART,
DHT22
Process Data:
Convert to PPM, Parse PM
values
Calculate & Classify AQI
Update LCD:
Show PM2.5, AQI,
Temperature
AQI
Hazardous?
Yes
No
Activate Buzzer
Deactivate Buzzer

Arduino Code:

```
// Arduino Uno sketch
#include <SoftwareSerial.h>
#include <PMS.h>            // include PMS library (install via Library Manager)
#include <ArduinoJson.h>    // for JSON formatting (optional but convenient)

#define MQ_PIN A0
#define BUZZER_PIN 8

SoftwareSerial pmsSerial(2, 3); // RX, TX  -> PMS5003 TX->2
PMS pms(pmsSerial);
PMS::DATA pmsData;

unsigned long lastSend = 0;
const unsigned long sendInterval = 5000; // ms

void setup() {
  Serial.begin(115200);       // USB Serial to Raspberry Pi
  pmsSerial.begin(9600);      // PMS5003 default baud
  pinMode(BUZZER_PIN, OUTPUT);
  digitalWrite(BUZZER_PIN, LOW);
  delay(1000);
  // wake PMS if needed:
  pms.wake();
}

float readMQ135() {
  int raw = analogRead(MQ_PIN); // 0-1023
  // Convert raw to approximate voltage (0-5V)
  float voltage = raw * (5.0 / 1023.0);
  // Simple mapping to "AQ index" or ppm — placeholder: calibrate properly
  // Many MQ135 modules require Ro/Rs calibration. We'll send raw ADC and voltage.
  return voltage;
}

void loop() {
  unsigned long now = millis();
  if (now - lastSend >= sendInterval) {
    lastSend = now;

    // Read PMS5003 (trigger a read)
    if (pms.read(&pmsData)) {
```

```
    // successful read
  } else {
    // failed read or no update yet
  }

  float mqVoltage = readMQ135();

  // Create a JSON object string manually (to keep memory small)
  // Example JSON: {"pm1":12,"pm2_5":20,"pm10":30,"mq_v":2.34}
  int pm1 = pmsData.PM_AE_UG_1_0;   // PM1.0 ug/m3 (auto-equivalent)
  int pm25 = pmsData.PM_AE_UG_2_5;  // PM2.5 ug/m3
  int pm10 = pmsData.PM_AE_UG_10;   // PM10 ug/m3

  // Optional: local alert logic
  if (pm25 > 100 || mqVoltage > 3.5) {
    digitalWrite(BUZZER_PIN, HIGH); // sound buzzer when very bad
  } else {
    digitalWrite(BUZZER_PIN, LOW);
  }

  // Send JSON over Serial (one line)
  Serial.print("{");
  Serial.print("\"pm1\":"); Serial.print(pm1); Serial.print(",");
  Serial.print("\"pm2_5\":"); Serial.print(pm25); Serial.print(",");
  Serial.print("\"pm10\":"); Serial.print(pm10); Serial.print(",");
  Serial.print("\"mq_v\":"); Serial.print(mqVoltage, 3);
  Serial.println("}");
 }

 // Let PMS run; short delay
 delay(100);
}
```

Explanation of the Code:

6.1.1 Flow of the Code

The Arduino sketch shown above forms the core logic for collecting, processing, and transmitting air quality data to the Raspberry Pi. It integrates readings from both the MQ135 gas sensor and the PMS5003 particulate matter sensor while providing real-time alerts through a buzzer. The following section describes the complete flow and functionality of the code.

1. Library Inclusion and Pin Configuration

The program begins by including three key libraries:

- SoftwareSerial.h – Allows creation of a secondary serial communication channel, essential for interfacing with the PMS5003 particulate sensor since the hardware serial port is used for communication with the Raspberry Pi.

- PMS.h – A dedicated library for the PMS5003 sensor that simplifies reading particulate matter values such as PM1.0, PM2.5, and PM10.

- ArduinoJson.h – Facilitates JSON-formatted data generation, making it easier for the Raspberry Pi to parse and process the readings.

Two pins are then defined:

- **MQ_PIN (A0): Analog input pin for the MQ135 sensor's output.**

- **BUZZER_PIN (8): Digital output pin used to drive the buzzer for alert notifications.**

Additionally, the PMS5003 is connected to the Arduino using pins 2 (RX) and 3 (TX) through a `SoftwareSerial` interface.

2. Initialization in `setup()`

During setup, three primary tasks are executed:

1. Serial Communication Setup

- `Serial.begin(115200)` initializes the main USB serial port for communication with the Raspberry Pi at 115200 baud rate.

- `pmsSerial.begin(9600)` initializes the PMS5003 communication channel at its default baud rate of 9600.

2. **Hardware Initialization**

    - **T**he buzzer pin is configured as an output and initially turned off using `digitalWrite(BUZZER_PIN, LOW)`.

    - The PMS sensor is "woken up" using the `pms.wake()` function to ensure it starts sending readings after power-up.

3. **Stabilization Delay**

    - **A one-second delay is added to allow hardware to stabilize before the main loop starts execution.**

3. Reading the MQ135 Sensor (`readMQ135()` Function)

This function handles the acquisition of analog data from the MQ135 gas sensor:

- The analog signal is read using `analogRead(MQ_PIN)`, producing a raw digital value between 0 and 1023.

- This value is converted to an approximate voltage using the formula:
  [
  \text{Voltage} = \text{Raw Value} \times \frac{5.0}{1023.0}
  ]

- This voltage acts as a relative indicator of air pollution levels (the higher the voltage, the poorer the air quality).

- In a calibrated setup, this voltage could be converted into precise parts-per-million (ppm) readings for gases like CO_2 or NH_3.

4. Main Program Logic (`loop()` Function)

The `loop()` function runs continuously, performing sensor reads and communication tasks at defined time intervals.

1. **Timed Execution Control**
    - The program ensures that sensor readings and data transmission occur every 5 seconds, controlled by `sendInterval = 5000` milliseconds.
    - This avoids excessive data transmission while maintaining real-time monitoring.
2. **Reading PMS5003 Data**
    - The PMS5003 sensor is polled using `pms.read(&pmsData)`.
    - Upon successful read, particulate matter concentrations are extracted:
        - `pmsData.PM_AE_UG_1_0` → PM1.0 concentration (μg/m³)
        - `pmsData.PM_AE_UG_2_5` → PM2.5 concentration (μg/m³)
        - `pmsData.PM_AE_UG_10` → PM10 concentration (μg/m³)
3. **Local Alert Logic**
    - After reading both sensors, the code checks if pollution exceeds safe thresholds:
        - If PM2.5 > 100 μg/m³ or MQ135 voltage > 3.5V, the buzzer is activated.
        - Otherwise, the buzzer remains off.
    - This feature provides immediate feedback about deteriorating air quality.
4. **Data Packaging and Transmission**
    - The collected readings are structured into a lightweight JSON object for easy parsing by the Raspberry Pi.
      Example output:

```
{"pm1":12,"pm2_5":20,"pm10":30,"mq_v":2.34}
```

- This JSON string is sent to the Raspberry Pi via USB serial using `Serial.print()` statements.

5. **Delay and Loop Continuation**
    - A short `delay(100)` at the end of each loop iteration ensures stable sensor communication and prevents rapid redundant polling.

5. Overall Workflow Summary

1. **Arduino initializes sensors and serial communication.**
2. **Every 5 seconds:**
    - Reads gas concentration from the MQ135 (via analog input).
    - Reads PM values (PM1.0, PM2.5, PM10) from PMS5003.
    - Activates the buzzer if pollutant levels exceed thresholds.
    - Transmits all readings to the Raspberry Pi in JSON format.
3. **The Raspberry Pi receives, stores, and visualizes these readings for continuous air quality monitoring.**

6. Key Functional Highlights

- Multi-sensor integration: Combines gas detection (MQ135) and particulate sensing (PMS5003).
- Data standardization: Uses JSON formatting for structured, machine-readable output.
- Local alert mechanism: Provides immediate environmental feedback through a buzzer.
- Efficient communication: Maintains steady data flow between Arduino and Raspberry Pi.

Chapter 07

Sensors

This chapter provides a detailed technical overview of the sensors employed in the air quality monitoring system. For each sensor, the operating principle, physical specifications, power ratings, and method of interfacing with the Raspberry Pi are described.

7.1 MQ135 Air Quality Gas Sensor

The MQ135 is a semiconductor-type gas sensor designed to detect a wide range of gases, making it suitable for general air quality monitoring. It is particularly sensitive to ammonia (NH3), nitrogen oxides (NOx), alcohol, benzene, smoke, and carbon dioxide (CO2).

7.1.1 Operating Principle

The sensing mechanism of the MQ135 is based on a chemoresistive material, specifically Tin Dioxide (SnO2). The core principle is as follows:

1. **Sensing Material:** The sensor contains a micro-tubular ceramic element coated with a layer of SnO2. In clean air, oxygen molecules are adsorbed onto the surface of the SnO2, trapping electrons from the material and creating a potential barrier that results in high electrical resistance.
2. **Gas Detection:** When pollutant gases are introduced, they react with the adsorbed oxygen molecules on the sensor's surface. This reaction releases the trapped electrons back into the SnO2, lowering the potential barrier and thereby decreasing the material's overall resistance.
3. **Signal Output:** This change in resistance is proportional to the concentration of the target gas. An external load resistor (RL) is used in a voltage divider circuit. As the sensor's resistance (RS) changes, the voltage across the load resistor (VRL) also changes, providing an analog voltage output that corresponds to the gas concentration.
4. **Heating Element:** The sensor includes an internal heating element made of Ni-Cr alloy, which is necessary to bring the SnO2 material to its optimal operating temperature (typically 100-200°C). This is why the sensor requires a pre-heating period of at least 24 hours for optimal performance and stabilization before accurate measurements can be taken.

7.1.2 Physical Dimensions and Pin Diagram

The MQ135 is commonly available as a module mounted on a small PCB for ease of use.

- **Physical Dimensions:** The module typically measures approximately 35 mm x 22 mm x 23 mm (L x W x H).
- **Pinout:** The module has a 4-pin header for connection:
    - **VCC:** Power supply input (5V).
    - **GND:** Ground connection.
    - **D0 (Digital Output):** Provides a HIGH or LOW signal based on a threshold set by an on-board potentiometer. This pin is not used in this project.
    - **A0 (Analog Output):** Provides the analog voltage signal proportional to gas concentration. This is the primary output used for this project.

7.1.3 Power Ratings and Technical Specifications

1. **Heater Voltage (VH):** 5.0V ± 0.1V AC or DC.
2. **Circuit Voltage (VC):** The sensor module operates at 5V DC.
3. **Heater Consumption (PH):** Approximately ≤950mW.
4. **Load Resistance (RL):** Adjustable via on-board potentiometer, but typically around 10 kΩ.
5. **Detection Range:** 10 – 1000 ppm for gases like ammonia and benzene.

7.1.4 Interfacing with Raspberry Pi

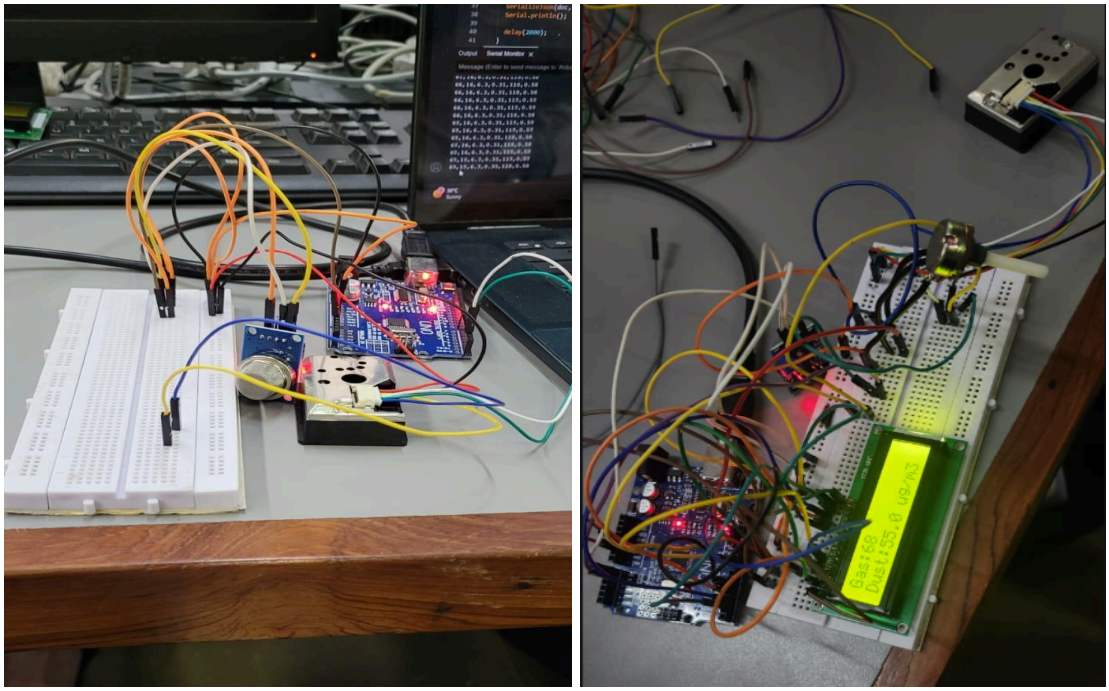
As the Raspberry Pi cannot process analog signals, the A0 pin of the MQ135 cannot be connected directly. An Arduino UNO serves as the necessary intermediary, acting as an Analog-to-Digital Converter (ADC).

- **Circuit:** The MQ135's VCC and GND pins are connected to the Arduino's 5V and GND pins, respectively. The sensor's A0 pin is connected to one of the Arduino's analog input pins (e.g., A0). The Arduino is then connected to the Raspberry Pi via a USB cable. The Arduino reads the analog voltage, converts it into a 10-bit digital value (0-1023), and sends this value over the serial connection to the Raspberry Pi, which can then read and process it.
- Preliminary connection of our sensors and raspberry pi is shown in the below image followed by explanation of the circuit.

7.1.5 Circuit Explanation

The above figure illustrates the preliminary circuit setup used for interfacing the MQ135 gas sensor with the Raspberry Pi through an Arduino UNO. The MQ135 sensor module is powered using a 5V supply from the Arduino, with the ground terminals of all components connected in common to ensure a stable reference. The sensor's analog output (A0) is connected to the Arduino's analog input pin (A0), enabling the Arduino to read the variable voltage corresponding to the gas concentration.

The Arduino functions as an Analog-to-Digital Converter (ADC), converting the analog voltage values from the MQ135 into 10-bit digital data (ranging from 0 to 1023). These readings are then transmitted serially to the Raspberry Pi through the USB connection. The Raspberry Pi receives and processes this data for further analysis, visualization, and storage.This arrangement allows continuous monitoring of air quality in real time while maintaining system flexibility. The setup also facilitates the integration of multiple sensors, as the same communication mechanism (Arduino-to-Raspberry Pi serial interface) can be extended to collect various environmental parameters simultaneously.



Chapter 08

Actuators and Displays

This chapter details the output components of the system: the devices that provide feedback to the user. It covers the operating principles, specifications, and interfacing methods for the local display and the audible alert mechanism.

8.1 16x2 I2C LCD Module

The 16x2 I2C LCD module is the primary visual interface for the system, providing real-time, human-readable information about the measured air quality parameters. It displays 16 characters per line across two lines.

8.1.1 Operating Principle

The module is a composite device consisting of two main parts: a standard character LCD and an I2C interface board.

- **Character LCD:** The display itself is typically based on the Hitachi HD44780 controller or a compatible equivalent. This controller requires a parallel interface (either 4-bit or 8-bit) to receive commands (like "clear display" or "move cursor") and data (the ASCII codes for characters to be displayed). Driving this directly would consume at least 6 GPIO pins from the Raspberry Pi.
- **I2C Interface Board:** To simplify wiring, an interface board is "piggy-backed" onto the LCD's 16-pin header. This board is built around a PCF8574 I/O expander chip. The PCF8574 acts as an I2C slave device. The Raspberry Pi, acting as the I2C master, sends commands and data over the two-wire I2C bus (SDA and SCL). The PCF8574 receives these serial commands and translates them into the parallel signals required by the HD44780 controller, managing the LCD's data and control lines. This elegant solution reduces the required GPIO pin count from six or more down to just two, which is a significant advantage in a pin-constrained project.

8.1.2 Physical Dimensions and Pin Diagram

- **Physical Dimensions:** The module's PCB is approximately 80 mm x 36 mm.
- **Pinout:** The I2C interface provides a simple 4-pin header for connection:
    - **GND:** Ground.
    - **VCC:** Power supply (5V).
    - **SDA (Serial Data):** The I2C data line.

- **SCL (Serial Clock):** The I2C clock line.

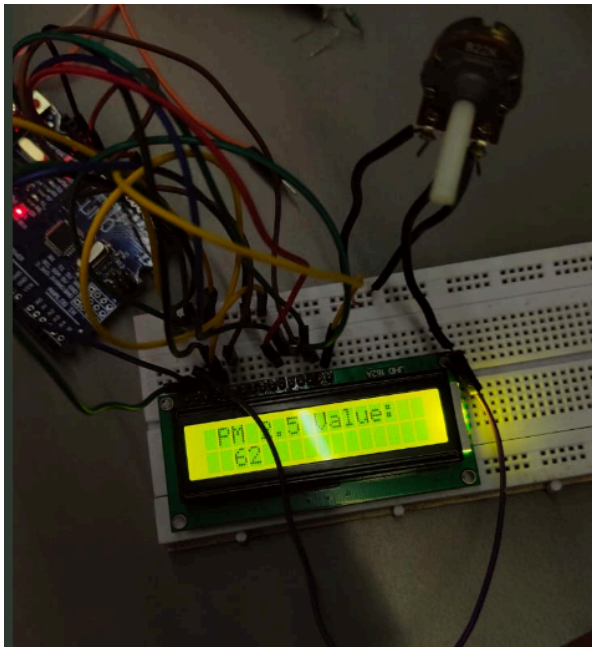
8.1.3 Power Ratings and Technical Specifications

- **Operating Voltage:** 5V DC.
- **Interface:** I2C (Inter-Integrated Circuit).
- **I2C Address:** The slave address is typically either 0x27 or 0x3F, depending on the specific I/O expander chip used on the interface board. The default address must be confirmed, often by running the `i2cdetect -y 1` command on the Raspberry Pi's terminal.
- **Display Format:** 16 characters x 2 lines.
- **Backlight:** Blue or green, controllable via software commands sent over I2C.
- **Contrast:** Adjustable via an on-board potentiometer.

8.1.4 Interfacing with Raspberry Pi

Interfacing the module with the Raspberry Pi is straightforward due to the I2C protocol.

- **Circuit:** The four pins of the LCD module (GND, VCC, SDA, SCL) are connected directly to the corresponding pins on the Raspberry Pi's GPIO header (GND, 5V, GPIO2/SDA, GPIO3/SCL). No other components are required for the connection. The Raspberry Pi's I2C interface must be enabled in the `raspi-config` utility.



Chapter 09

Details of Website Used

For data visualization and monitoring, the project integrates **Grafana**, an open-source analytics and monitoring platform. Grafana was chosen because it provides a clean, interactive interface for real-time sensor readings such as **Temperature**, **PPM**, **AQI Index**, and **Humidity**.

The Raspberry Pi continuously uploads the sensor data (from MQ135, PMS5003, and DHT11/DHT22) to a cloud database. Grafana connects to this data source through **Prometheus** using an HTTP API. The readings are automatically updated on dynamic dashboards, which allow users to view historical trends, compare multiple parameters, and analyze air quality patterns efficiently.

Technical Details

Parameter	Description
Platform	Grafana
Data source	Prometheus database(HTTP pull model)
Connection Protocol	HTTP
Display type	Time series graphs and gauge panels
Measured Parameters	Temperature (°C), Humidity (%), Gas Concentration (PPM), Air Quality Index (AQI)
Update Frequency	Every 5 seconds (as configured in Raspberry Pi script)

Working

1. The Raspberry Pi collects sensor data and hosts a lightweight Prometheus exporter (Python Flask server) on port 8000.

2. Prometheus pulls the data periodically and stores it in a time-series database.

3. Grafana queries Prometheus to render the visual graphs in real-time.

4. Users can access the Grafana dashboard remotely via a secure IP or Grafana Cloud link.

Dashboard Screenshot

Video Demonstration : [link1](#) , [link2](#)

Chapter 10

10.1 Protocols Used to connect Arduino and Raspberry Pi:

1.)Arduino → Raspberry Pi

- Protocol: Serial Communication **(UART)**
- How: Arduino sends sensor readings (from MQ135 & GP2Y1014AU0F) via USB cable to Raspberry Pi.
- Details:
    - Uses standard serial communication (`Serial.begin(9600)` in Arduino).
    - Raspberry Pi reads this using Python (`pyserial` library).

---

2. Raspberry Pi → Prometheus / Grafana

There are two possible paths, depending on setup:

a) Using Prometheus

- Protocol: **HTTP / REST** (Pull Model)
- **How:** Raspberry Pi runs a small **Prometheus exporter** (a Python Flask app or Node exporter) that serves sensor data at an HTTP endpoint like `http://<raspberrypi-ip>:8000/metrics`.
- **Prometheus** periodically pulls data from this endpoint using **HTTP GET**.

b) Using Grafana (direct or via Prometheus)

- Protocol: **HTTP** (Dashboard Visualization)
- Grafana connects to Prometheus via **HTTP API** to fetch data for visualization.

- If using Grafana Cloud, Raspberry Pi pushes data via **HTTPS** (secure HTTP).

10.2 Difference between wi-fi , bluetooth and zigbee:

Parameter	Bluetooth	Wi-Fi (IEEE 802.11)	Zigbee (IEEE 802.15.4)
Primary Purpose	Short-range wireless communication between personal devices (e.g., headsets, smartphones, wearables).	High-speed data networking and Internet connectivity for devices (computers, routers, IoT gateways).	Low-power, low-data-rate communication for IoT and sensor networks.
Standard / IEEE	IEEE 802.15.1	IEEE 802.11 (a/b/g/n/ac/ax)	IEEE 802.15.4
Frequency Band	2.4 GHz ISM band (Globally available)	2.4 GHz, 5 GHz (some versions support 6 GHz Wi-Fi 6E)	2.4 GHz (global), 868 MHz (Europe), 915 MHz (USA)
Data Rate	Up to 3 Mbps (Classic), ~2 Mbps (Bluetooth 5 LE)	11 Mbps (802.11b) to >9.6 Gbps (Wi-Fi 6)	20 kbps (868 MHz), 40 kbps (915 MHz), 250 kbps (2.4 GHz)
Range	10 m (Class 2), up to 100 m (Class 1 devices)	~50 m indoors, up to 100 m outdoors (depending on router power)	Typically 10–100 m (can extend via mesh networks)
Topology	Point-to-point, piconet (1 master, up to 7 slaves), scatternet	Star or infrastructure mode (Access Point with multiple clients); ad-hoc possible	Mesh, star, or tree topology; designed for many low-power nodes
Power Consumption	Low to moderate; optimized in BLE (Bluetooth Low Energy)	High; continuous power draw suitable for mains-powered devices	Very low; optimized for battery-powered and always-on devices
Security	128-bit AES encryption,	WPA2/WPA3 (AES-based), strong	128-bit AES encryption (mandatory), secure

	authentication, pairing mechanisms	encryption and authentication	key management for networks
Latency	Low (<10 ms for BLE)	Moderate (1–100 ms, depending on congestion)	Low to moderate (~30 ms typical)
Network Capacity	8 active devices per piconet (Classic), thousands with BLE Mesh	200+ devices per Access Point (depends on implementation)	65,000+ nodes in a Zigbee mesh network
Throughput Efficiency	Moderate (suitable for small data packets)	High (supports streaming, HD video, large file transfers)	Low (optimized for periodic sensor data, not streaming)
Device Cost	Low	Moderate to high	Very low
Typical Applications	Audio devices, health wearables, smart watches, file sharing, IoT sensors (BLE)	Internet access, video streaming, laptops, smartphones, routers, IoT gateways	Home automation (lights, sensors, thermostats), industrial control, smart energy meters
Advantages	Ubiquitous, easy pairing, supported by most consumer devices	High data rate, wide coverage, robust ecosystem	Extremely low power, scalable, self-healing mesh, ideal for IoT
Limitations	Limited range, moderate interference in crowded 2.4 GHz band	High power consumption, complex setup, not ideal for battery-based devices	Low throughput, small packet size, interoperability issues in early devices
Interference Susceptibility	High (shares 2.4 GHz with Wi-Fi and microwaves)	High (crowded 2.4 GHz spectrum)	Low to moderate (frequency agility helps avoid congestion)
Ideal Use Case	Short-range personal communication between nearby devices	High-bandwidth data transfer and Internet access	Low-data, low-power IoT networks requiring scalability and reliability

Chapter 11

External Tools Used :

1. Grafana-

Grafana is an open-source analytics and visualization platform used for monitoring real-time data from IoT systems. In this project, Grafana is used to visualize the environmental data collected from sensors (MQ135, GP2Y1014AU0F, and DHT11) through clear and interactive dashboards.

Grafana enables the representation of sensor readings such as gas concentration, particulate matter levels (PM2.5), temperature, and humidity in the form of graphs, gauges, and time-series charts, providing an intuitive understanding of air quality variations over time.

Grafana connects to Prometheus as its data source and automatically refreshes the displayed information, ensuring real-time updates. It also supports setting threshold-based alerts, allowing the system to notify users when air quality parameters exceed safe limits.

Key Features Used:

- Real-time visualization of sensor data.
- Custom dashboard design for air quality parameters.
- Integration with Prometheus for live data streaming.
- Alert configuration for threshold breaches.

Grafana's flexibility, open-source nature, and powerful visualization capabilities make it an ideal tool for IoT-based environmental monitoring applications like this project.

2. Prometheus-

Prometheus is an open-source time-series database and monitoring tool designed for collecting, storing, and querying metrics data efficiently. In this project, Prometheus is used to collect and store real-time sensor data transmitted from the Raspberry Pi.

The Raspberry Pi runs a lightweight data exporter or script that periodically pushes air quality readings (gas concentration, particulate matter, humidity, and temperature) to Prometheus using the HTTP-based metrics endpoint. Prometheus then records this data with timestamps, allowing for accurate trend analysis and historical comparisons.

Key Features Used:

- Time-series data collection from the Raspberry Pi.
- Efficient data storage and retrieval for real-time monitoring.
- Seamless integration with Grafana for dashboard visualization.
- Support for PromQL (Prometheus Query Language) for data filtering and analysis.

Prometheus provides a robust backend for data monitoring and acts as a data source for Grafana, ensuring smooth communication between the IoT system and the visualization layer.

Chapter 12

1. Arduino Code:

```
#include <LiquidCrystal.h>
#include "DHT.h"

#define DHTPIN 7
#define DHTTYPE DHT11

DHT dht(DHTPIN, DHTTYPE);

// LCD Pins (RS, EN, D4, D5, D6, D7)
LiquidCrystal lcd(12, 11, 5, 4, 3, 2);

int mq135Pin = A0;     // MQ135 gas sensor
int dustLEDPin = 8;    // LED control pin for GP2Y1014
int dustAoutPin = A1; // Analog output from GP2Y1014

float VoMeasured = 0;
float calcVoltage = 0;
float dustDensity = 0;

void setup() {
  pinMode(dustLEDPin, OUTPUT);
```

```
  lcd.begin(16, 4);
  Serial.begin(9600);
  dht.begin();
  lcd.print("Air Quality Init");
  delay(2000);
  lcd.clear();

}

void loop() {
  // --- MQ135 Reading ---
  int gasValue = analogRead(mq135Pin);

  // --- GP2Y1014 Dust Sensor Reading ---
  digitalWrite(dustLEDPin, LOW);   // Turn on the LED
  delayMicroseconds(280);
  VoMeasured = analogRead(dustAoutPin);
  delayMicroseconds(40);
  digitalWrite(dustLEDPin, HIGH);  // Turn off the LED
  delayMicroseconds(9680);

  calcVoltage = VoMeasured * (5.0 / 1024.0);  // Convert ADC to voltage
  dustDensity = VoMeasured;  // From Sharp datasheet

  // Limit dust density to realistic range
  // if (dustDensity < 0) dustDensity = 0;
  //Dht Sensor
  float h = dht.readHumidity();
  float t = dht.readTemperature();

  if (isnan(h) || isnan(t)) {
    Serial.println("Failed to read from DHT sensor!");
    return;
  }

  // --- Serial Output ---
  Serial.print("Gas: ");
  Serial.print(gasValue);
  Serial.print(" | Dust: ");
  Serial.print(dustDensity, 1);
```

```
  Serial.print(" ug/m3 | Hum: ");
  Serial.print(h, 1);
  Serial.print(" % | Temp: ");
  Serial.print(t, 1);
  Serial.println(" C");

  // --- LCD Output ---
  lcd.clear();
  lcd.setCursor(0, 0);
  lcd.print("Gas:");
  lcd.print(gasValue,1);
  lcd.print(" AQI");
  lcd.setCursor(0, 1);
  lcd.print("Dust:");
  lcd.print(dustDensity,1);
  lcd.print(" ug/m3");

  lcd.setCursor(0, 2);
  lcd.print("Hum: ");
  lcd.print(h);
  lcd.print("% " );
  lcd.setCursor(0, 3);
  lcd.print("Temp:");
  lcd.print(t,1);
  lcd.print((char)223);
  lcd.println("C      ");
  delay(2000);
}
```

2. **Raspberry PI Python Code :**

```
from flask import Flask, Response
from prometheus_client import Gauge, generate_latest, CONTENT_TYPE_LATEST
import serial
import time
import threading

# === Prometheus metrics ===
```

```
gas_gauge = Gauge('mq135_gas_value', 'Gas sensor reading from MQ135')
dust_gauge = Gauge('gp2y1014_dust_density', 'Dust density (ug/m3) from GP2Y1014')
humidity_gauge = Gauge('dht11_humidity', 'Humidity (%) from DHT11')
temperature_gauge = Gauge('dht11_temperature', 'Temperature (°C) from DHT11')

# === Flask setup ===
app = Flask(__name__)

# === Arduino serial connection ===
ser = serial.Serial('/dev/ttyACM0', 9600, timeout=1)
time.sleep(2)

def read_serial_data():
    while True:
        try:
            line = ser.readline().decode('utf-8').strip()
            if line:
                print("Received:", line)
                # Expected format:
                # Gas: 220 | Dust: 40.2 ug/m3 | Hum: 48.5 % | Temp: 26.9 C
                parts = line.split('|')
                gas = float(parts[0].replace("Gas:", "").strip())
                dust = float(parts[1].replace("Dust:", "").replace("ug/m3", "").strip())
                hum = float(parts[2].replace("Hum:", "").replace("%", "").strip())
                temp = float(parts[3].replace("Temp:", "").replace("C", "").strip())

                # Update Prometheus metrics
                gas_gauge.set(gas)
                dust_gauge.set(dust)
                humidity_gauge.set(hum)
                temperature_gauge.set(temp)

        except Exception as e:
            print("Error:", e)
        time.sleep(2)

@app.route("/metrics")
def metrics():
    return Response(generate_latest(), mimetype=CONTENT_TYPE_LATEST)
```

```
if __name__ == "__main__":
    thread = threading.Thread(target=read_serial_data)
    thread.daemon = True
    thread.start()
    app.run(host="0.0.0.0", port=8000)
```

For connecting we used Grafana Cloud where we generated our API key and username with which we created a Yaml file to create remote endpoints.

Chapter 13

Project Summary and Timeline

This project demonstrates an **IoT-based Air Quality Monitoring and Alerting System** designed to measure environmental parameters and visualize them in real time.

The system combines sensors for particulate matter (PM2.5, PM10), temperature, humidity, and gas concentration with a Raspberry Pi acting as the central controller. Data is transmitted to Grafana through Prometheus for visual analysis.

Timeline of Work

Phase	Duration	Description
Project Ideation	Week 1	Identification of problem: need for affordable, real-time air quality monitoring.
Component Research	Week 2 - 3	Study and procurement of sensors (MQ135, PMS5003, DHT11) and Raspberry Pi.
Circuit Design	Week 4	Schematic and wiring between sensors, Arduino UNO (ADC), and Raspberry Pi.
Coding and Integration	Week 5 – 7	Development of Arduino sketch and Python scripts for

		data collection and communication.
Testing and Calibration	Week 8	Validation of readings and threshold adjustments.
Visualization Setup	Week 9	Configuration of Prometheus and Grafana dashboard.
Final Demonstration	Week 10	Complete system tested and deployed for real-time monitoring.

Working Description

In real-world operation:

- The **MQ135** detects harmful gases.
- The **PMS5003** measures particulate matter.
- The **DHT11/DHT22** captures temperature and humidity.
- The **Arduino UNO** converts analog data and sends it to the **Raspberry Pi**, which calculates the AQI value and triggers a buzzer when hazardous levels are reached.
- The **Grafana dashboard** provides continuous visual updates to users anywhere through an Internet connection.

Future Scope

The system can be enhanced in the following ways:

1. **Solar-powered Operation** – Integration of solar panels and battery backup for off-grid use.
2. **Mobile App Notification System** – Real-time alerts through mobile applications or SMS when AQI exceeds safe limits.
3. **AI-based Prediction** – Using machine learning to forecast pollution trends based on historical data.

Chapter 14

Appendix A

References:

- A. Kumar, S. Tiwari, and R. Singh, “IoT based real-time air quality monitoring system using sensors and cloud computing,” IEEE Internet of Things Journal, vol. 7, no. 9, pp. 8942–8950, Sep. 2020.
- M. A. Alvear, E. Natalizio, and C. Calafate, “Smart IoT-based air pollution monitoring systems for urban environments,” IEEE Sensors Journal, vol. 21, no. 14, pp. 15863–15872, Jul. 2021.
- P. Saini and A. Sharma, “Air quality prediction and classification using machine learning techniques,” IEEE Access, vol. 9, pp. 103106–103118, 2021.
- T. R. Gadekar, R. Patil, and S. Patil, “Design and implementation of IoT-based air quality monitoring and alerting system,” in Proc. IEEE Int. Conf. on Communication and Electronics Systems (ICCES), Coimbatore, India, 2022, pp. 1234–1238.
- M. K. Gupta and A. Chauhan, “Integration of IoT and AI for real-time air pollution monitoring and prediction,” IEEE Transactions on Instrumentation and Measurement, vol. 72, pp. 1–10, 2023.
- World Health Organization, “Ambient (outdoor) air pollution,” WHO Fact Sheet, Geneva, Switzerland, Apr. 2023. [Online]. Available: Three Real-Life Implementation Challenges and Their Resolutions

MQ135 Semiconductor Sensor for Air Quality

Profile

Sensitive material of MQ135 gas sensor is SnO_2, which with lower conductivity in clean air. When target pollution gas exists, the sensor's conductivity gets higher along with the gas concentration rising. Users can convert the change of conductivity to correspond output signal of gas concentration through a simple circuit.

MQ135 gas sensor has high sensitivity to ammonia gas, sulfide, benzene series steam, also can monitor smoke and other toxic gases well. It can detect kinds of toxic gases and is a kind of low-cost sensor for kinds of applications.

Features

It has good sensitivity to toxic gas in wide range, and has advantages such as long lifespan, low cost and simple drive circuit &etc.

Main Applications

It is widely used in domestic gas alarm, industrial gas alarm and portable gas detector.

Technical Parameters Stable.1

Model			MQ135
Sensor Type			Semiconductor
Standard Encapsulation			Bakelite, Metal cap
Target Gas			ammonia gas, sulfide, benzene series steam
Detection range			10~1000ppm(ammonia gas, toluene, hydrogen, smoke)
Standard Circuit Conditions	Loop Voltage	V_c	≤24V DC
	Heater Voltage	V_H	5.0V±0.1V AC or DC
	Load Resistance	R_L	Adjustable
Sensor character under standard test conditions	Heater Resistance	R_H	29Ω±3Ω (room tem.)
	Heater consumption	P_H	≤950mW
	Sensitivity	S	Rs(in air)/Rs(in 400ppm H_2)≥5
	Output Voltage	Vs	2.0V~4.0V（in 400ppm H_2）
	Concentration Slope	α	≤0.6(R_{400ppm}/R_{100ppm} H_2)
Standard test conditions	Tem. Humidity		20℃±2℃; 55%±5%RH
	Standard test circuit		Vc:5.0V±0.1V; V_H: 5.0V±0.1V
	Preheat time		Over 48 hours

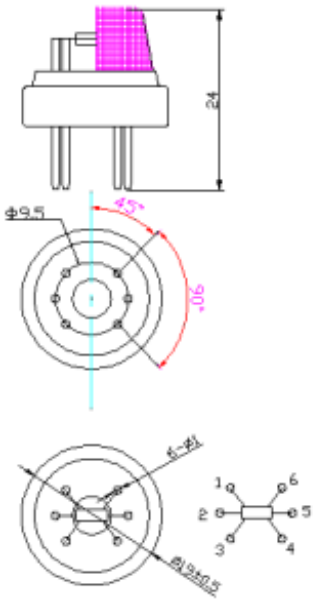


Fig1.Sensor Structure

Unit: mm

NOTE: Output voltage (Vs) is V_{RL} in test environment.

PMS2005 Sensor:

Plantower PMS5003 is a laser dust sensor. Sensor uses laser light scattering principle to measure value of dust particles suspended in the air. Sensor provides precise and reliable reading of PM2.5 value.

Specification:

- Supply voltage: from 4.5 V to 5.5 V
- Power consumption (work): below 100 mA
- Power consumption (standby): below 200 µA
- Sensitivity:
  - 50% - 0.3 µm
  - 98% - 0.5 µm and larger
- Resolution: 1 µg/m³
- Work temperature: od -10 °C do 60 °C
- Humidity (work): 0-99%
- Size: 50 x 38 x 21 mm

Set includes:

- PMS5003 sensor (covered with protective film)
- Cable

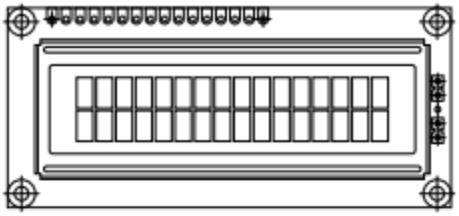
Pinout:

Pin 1 from inside. Pin 8 on the outside. 8-pin connector (1.25mm raster).

Pin	Function	Description	Remarks
1	**VCC**	Supply voltage 5V	4.5 - 5.5V
2	**GND**	Ground	
3	**SET**	HIGH or SUSPENDED - work mode LOW - sleep mode	`3.3V logic`
4	**RXD**	UART/TTL data recieve	`3.3V logic`
5	**TXD**	UART/TTL data transmit	`3.3V logic`
6	**Reset**	LOW to reset	`3.3V logic`
7	**NC**	Not connected	
8	**NC**	Not connected	

LCD 16X2: A liquid crystal display module that can show 16 characters per line on two lines, for a total of 32 characters at a time

16 x 2 Character LCD



FEATURES

- Type: Character
- Display format: 16 x 2 characters
- Built-in controller: ST 7066 (or equivalent)
- Duty cycle: 1/16
- 5 x 8 dots includes cursor
- + 5 V power supply
- LED can be driven by pin 1, pin 2, or A and K
- N.V. optional for + 3 V power supply
- Optional: Smaller character size (2.95 mm x 4.35 mm)
- Material categorization: For definitions of compliance please see www.vishay.com/doc?99912

RoHS COMPLIANT

MECHANICAL DATA		
ITEM	STANDARD VALUE	UNIT
Module Dimension	80.0 x 36.0 x 13.2 (max.)	mm
Viewing Area	66.0 x 16.0	
Dot Size	0.55 x 0.65	
Dot Pitch	0.60 x 0.70	
Mounting Hole	75.0 x 31.0	
Character Size	2.95 x 5.55	

ABSOLUTE MAXIMUM RATINGS					
ITEM	SYMBOL	STANDARD VALUE			UNIT
		MIN.	TYP.	MAX.	
Power Supply	V_{DD} to V_{SS}	- 0.3	-	13	V
Input Voltage	V_I	V_{SS}	-	V_{DD}	

Note

- V_{SS} = 0 V, V_{DD} = 5.0 V

ELECTRICAL CHARACTERISTICS						
ITEM	SYMBOL	CONDITION	STANDARD VALUE			UNIT
			MIN.	TYP.	MAX.	
Input Voltage	V_{DD}	V_{DD} = + 5 V	4.5	5.0	5.5	V
Supply Current	I_{DD}	V_{DD} = + 5 V	1.0	1.2	1.5	mA
Recommended LC Driving Voltage for Normal Temperature Version Module	V_{DD} to V_0	- 20 °C	-	-	5.2	V
		0 °C	-	-	-	
		25 °C	-	3.7	-	
		50 °C	-	-	-	
		70 °C	3.1	-	-	
LED Forward Voltage	V_F	25 °C	-	4.2	4.6	V
LED Forward Current - Array	I_F	25 °C	-	100	-	mA
LED Forward Current - Edge			-	20	40	
EL Power Supply Current	I_{EL}	V_{EL} = 110 V_{AC}, 400 Hz	-	-	5.0	mA

DISPLAY CHARACTER ADDRESS CODE																
Display Position	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
DD RAM Address	00	01	02	03	04	05	06	07	08	09	0A	0B	0C	0D	0E	0F
DD RAM Address	40	41	42	43	44	45	46	47	48	49	4A	4B	4C	4D	4E	4F

Appendix B

Programming Review: Comparing C and Python with Java, and Rust

Purpose

A concise comparative review of four widely used programming languages — **C**, **Python**, **Java**, and **Rust** — intended for Appendix B of your report. Each language is compared across design goals, syntax style, memory model, performance, concurrency, typical use-cases, strengths, and weaknesses.

1. Language Overviews

C :

- **Design goal:** Minimal, portable systems programming language with close-to-hardware control.
- **Typing:** Static, weak (no automatic safety checks).
- **Execution model:** Compiled to native machine code.

Python :

- **Design goal:** High-level, readable, productive language focused on developer ergonomics.
- **Typing:** Dynamic, strong (runtime type checking).
- **Execution model:** Interpreted (commonly CPython) or compiled to bytecode; many implementations exist.

Java :

- **Design goal:** Portable, object-oriented language with a managed runtime (JVM) and strong standard libraries.
- **Typing:** Static, strong.

- **Execution model:** Compiled to JVM bytecode, executed on Java Virtual Machine.

Rust :

- **Design goal:** Systems programming with memory safety guarantees and high performance without a garbage collector.
- **Typing:** Static, strong.
- **Execution model:** Compiled to native machine code.

2. Key Characteristics (By Topic)

Syntax & Readability

- **C:** Compact but minimal abstractions; can become terse. Manual memory management (malloc/free) shapes readability around resource handling.
- **Python:** Very readable and concise; significant whitespace and expressive standard library increase productivity and reduce boilerplate.
- **Java:** Verbose compared to Python; strong conventions and explicit object-oriented structure; clear but more boilerplate.
- **Rust:** More verbose than Python; modern syntax with expressive constructs (pattern matching, traits) — readable once accustomed.

Memory Management

- **C:** Manual; programmer responsibility; powerful but error-prone (use-after-free, leaks, buffer overflows).
- **Python:** Automatic garbage collection; programmer seldom handles memory directly, trading performance for convenience.
- **Java:** Garbage-collected via JVM; deterministic-ish pause behavior depends on GC algorithm and tuning.

- **Rust:** Ownership/borrowing model enforced at compile-time, enabling memory safety without a runtime GC.

Performance

- **C:** Excellent — minimal runtime overhead; often used where maximum performance and small binary size are required.

- **Python:** Slower due to interpretation and dynamic typing; speed can be mitigated via C extensions, JITs (PyPy), or vectorized libraries.

- **Java:** High-performance JIT compiled code; often competitive with native code for long-running workloads.

- **Rust:** Comparable to or sometimes better than C due to modern optimizations and zero-cost abstractions.

Concurrency & Parallelism

- **C:** Concurrency via OS threads, pthreads, or third-party libraries; no language-level safety for data races.

- **Python:** Concurrency through threads (GIL limits CPU-bound parallelism) and multiprocessing; async IO (async/await) for scalable IO-bound programs.

- **Java:** Mature concurrency primitives (threads, executors, concurrency utilities); JVM provides robust tooling for concurrent applications.

- **Rust:** Concurrency enforced by the type system (ownership/borrowing) to prevent data races at compile time; strong async ecosystem.

Tooling & Ecosystem

- **C:** Mature compilers (gcc, clang), debuggers (gdb), profilers; large body of existing libs and platform interfaces.

- **Python:** Rich package ecosystem (PyPI), interactive REPL, tooling (linters, formatters, test frameworks) and strong data/ML ecosystem.

- **Java:** Extensive ecosystem (Maven/Gradle), enterprise libraries, strong IDE support (IntelliJ, Eclipse), and robust frameworks (Spring).

- **Rust:** Fast-growing ecosystem (Cargo package manager), good tooling (rustc, rustfmt, clippy), increasing library support.

3. Typical Use-Cases

- **C:** Operating systems, embedded systems, device drivers, performance-critical libraries, systems utilities.

- **Python:** Scripting, automation, web backends, data science, machine learning, prototyping, glue code.

- **Java:** Enterprise backend systems, Android apps (historically), large-scale servers, cross-platform applications.

- **Rust:** Systems programming, safe concurrent services, performance-critical components where memory safety matters (e.g., networking, cryptography).

4. Strengths & Weaknesses

C :

- **Strengths:** Performance, predictable behavior, small runtime, universal platform support.

- **Weaknesses:** Manual memory errors, limited abstractions, harder to write safe concurrent code.

Python :

- **Strengths:** Rapid development, readability, huge ecosystem (data science, web), gentle learning curve.

- **Weaknesses:** Runtime performance, dynamic typing may hide bugs until runtime, less suitable for low-level systems work.

Java :

- **Strengths:** Portability, robust standard library, strong tooling and ecosystem, good concurrency support.
- **Weaknesses:** Verbose, requires JVM (startup cost), memory overhead from managed runtime.

Rust :

- **Strengths:** Memory safety without GC, high performance, strong concurrency safety, modern tooling.
- **Weaknesses:** Steeper learning curve (ownership model), smaller ecosystem than older languages (though growing).

5. Short Code Examples — "Hello, World!"

C

```c
#include <stdio.h>
int main(void) {
    printf("Hello, World!\n");
    return 0;
}
```

Python

```python
print("Hello, World!")
```

Java

```java
public class Hello {
    public static void main(String[] args) {
        System.out.println("Hello, World!");
    }
```

```
}
```

Rust

```rust
fn main() {
    println!("Hello, World!");
}
```

6. Conclusion

C and Rust occupy the lower-level/systems niche where performance and control matter; Rust improves safety at compile time while keeping performance. Python emphasizes fast development and readability at the cost of raw speed. Java offers a middle path with a managed runtime and strong enterprise support. The best language depends on the problem domain, team expertise, deployment constraints, and performance/safety trade-offs.

7. Suggested further reading (non-exhaustive)

- Language official documentation and tutorials (C standard docs, Python.org, Oracle Java docs, Rust Book).
- Community-written comparisons and performance benchmarks for specific domains.

Appendix C

Trouble-shooting

During the project's development, we encountered several technical challenges. This section details one significant problem, outlining the diagnostic process and the eventual solution.

Problem: Inaccurate and Static Sensor Readings

During the initial integration testing, we faced a critical issue where the dust sensor (a PM2.5 particle sensor) was returning static, nonsensical data. The main application was running, and the LCD display would correctly show "AQI: 0" or "AQI: 999" (our error value), but the value would not change, even when we introduced dust particles near the sensor.

Troubleshooting and Debugging Process

1. **Physical Layer Verification:** Our first step was to assume a hardware fault. We powered down the Raspberry Pi and re-checked all wiring connections between the sensor and the Pi's GPIO header. We used a multimeter to confirm that the sensor was receiving the correct voltage (5V) and that the Ground (GND) connection was solid.

2. **Code Isolation:** The next step was to check a bug in our main application. We wrote a separate function to import the sensor's library (e.g., `pms5003`), initialize the serial connection, and print the raw data to the console. This isolated the problem to the sensor-Pi interface. When we ran this test script, it also failed, either hanging or printing empty data, confirming the issue was not with our AQI calculation logic.

3. **System-Level Configuration Check:** Since the hardware and the isolated code seemed correct, we investigated the Raspberry Pi's configuration. PM2.5 sensors often communicate over a serial (UART) interface. On a Raspberry Pi, the GPIO pins for serial communication can be used for two purposes: a general-purpose serial interface *or* a system-level Linux serial console for debugging the Pi itself.

Solution

1. We ran the Raspberry Pi configuration tool from the terminal: **sudo raspi-config.**
2. We navigated to **Interface Options > Serial Port.**
3. We discovered that the "serial console" was **Enabled**, and the "serial port hardware" was **Disabled**. This was the source of the conflict. The operating system was "holding" the port, preventing our Python script from accessing it.
4. We applied the correct settings, after saving these changes and rebooting the Raspberry Pi, we ran our minimal `test_sensor.py` script again. This time, the console was displaying the valid, changing data packets from the sensor.
5. We then ran our main project application, and it worked perfectly. The sensor data was read correctly, the AQI was calculated, and the live index value was displayed on the LCD and successfully prepared for transmission over Wi-Fi.

Appendix D

Real-Life Mounting of the System

In a real-world deployment, the air-quality monitoring unit would be installed **outdoors at a height of 2–3 meters**, ensuring unobstructed air circulation and representative readings.
To protect the device:

- **Weather Protection:** The sensors and Raspberry Pi are enclosed in an **IP-rated waterproof polycarbonate housing** with small vent holes shielded by mesh to allow airflow but prevent dust or water ingress.
- **Damage Prevention:** The unit is mounted on a **metal pole with anti-vandal screws** and optionally placed inside fenced areas such as street-light poles or school premises.
- **Power Supply:** A **5 V / 2.5 A DC adapter** or **solar-battery system** ensures continuous operation.
- **Internet Connectivity:** Wi-Fi from nearby infrastructure or a **4G LTE dongle** provides reliable data transmission to the cloud.

Regular maintenance would include cleaning the vent holes, verifying sensor calibration, and checking power integrity every 3 months.

Appendix E

Real-Life Scenarios

Challenges and Proposed Solutions

1. **Challenge 1: Harsh Environmental Conditions**
   Problem*:* Dust, rain, and heat may degrade sensor performance.
   Solution*:* Use weatherproof enclosures with breathable membranes and install temperature-compensated calibration routines.

2. **Challenge 2: Network Connectivity Issues**
   Problem*:* Unstable Wi-Fi or loss of Internet could interrupt data uploads.
   Solution*:* Implement local data buffering on the Raspberry Pi using SQLite or CSV storage, which automatically syncs to the cloud once the connection is restored.

3. **Challenge 3: Power Supply Reliability**
   Problem*:* Power cuts could disable continuous monitoring.
   Solution*:* Include a rechargeable battery or solar panel backup with automatic switching to maintain uptime during outages.