Ahmedabad
University

Project Report-5
Group-8
Deadline: 6/11/2025

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

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

Chapter 9

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.

9.1 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.

9.1.1 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 32GB 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.

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, authentication, pairing mechanisms	WPA2/WPA3 (AES-based), strong encryption and authentication	128-bit AES encryption (mandatory), secure 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

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/C++

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

Python

Python

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

Java

Java

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

Rust

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.