3D Printed Robot

Wireless Communication and Sensing

EE3L11
Bachelor Graduation Project

Weikai Chen & Guangran Ran

Delft University of Technology

TU Delft

3D Printed Robot

Wireless Communication and Sensing

by

Weikai Chen & Guangran Ran

Instructor:	Dr. J. (Jianning) Dong
Teaching Assistant:	Ir. S. (Sachin) Yadav
Project Duration:	April, 2025 - June, 2025
Faculty:	Faculty of Electrical Engineering, Delft

Cover: Canadarm 2 Robotic Arm Grapples SpaceX Dragon by NASA under CC BY-NC 2.0 (Modified)
Style: TU Delft Report Style, with modifications by Daan Zwaneveld

TU Delft

Preface

This thesis has been written as part of the Bachelor of Science in Electrical Engineering at Delft University of Technology. It was conducted as a collaborative group project involving six students, divided into three subgroups: wireless communication and sensing, power supply, and drive control. This thesis specifically presents the work carried out by the wireless communication and sensing team.

The project was conducted within the framework of the GEMS Erasmus+, which aims to promote innovative, inclusive, and accessible learning in mechatronics education for students. As part of this initiative, we made use of custom-designed PCBs, those PCBs served as the foundation for the design and implementation of a wireless communication system, enabling data transmission between the hardware the host device.

Throughout this project, our understanding of wireless communication, embedded systems, and sensor integration has grown. We have also gained valuable experience in team-based development and and pratical system design.

We would like to express our sincere gratitude toward our supervisor, Prof.Jianning Dong, for his expert guidance, and to Dr.Sachin Yadav for his support and technical assistance throughout the course of this project.

We also want to thank our fellow group members for their contributions and collaboration: Parama Parama Fawwaz and Delan Duijne for the power supply subgroup, and Joshua van der Geize and Robin Appel from the drive control subgroup. Their efforts were essential to the overall success of the project.

Weikai Chen & Guangran Ran
Delft, June 2025

Abstract

This project reports the design and validation of a wheeled mobile robot that collects spatially distributed environmental data inside a building. Built on the four-board GEMS stack—Sapphire (communication), Ruby (sensing), Diamond (motor control), and Emerald (power), the prototype integrates a CAN bus backbone, an I2C sensor backplane, ultrasonic obstacle detection, and a Wi-Fi web interface. Responsibilities were divided among three subgroups. The wireless team implemented the CAN network, web server, and finite-state machine for autonomous navigation. The mechanical team designed and printed the chassis; they also developed battery management and voltage regulation. The motor control group implemented the PI control.

Integration tests show that the system satisfies every “must-have” requirement in the Programme of Requirements: CAN frame loss remained below 1 % during a five-minute run, Wi-Fi throughput exceeded 1 Mbps in the range of ten meters, ultrasonic sensors detected obstacles from 10 cm to 30 cm and temperature and humidity data were logged at 2 Hz with millisecond precision. However, late delivery of prefabricated cabling forced a temporary hand-wired CAN harness, leaving room for mechanical refinement.

Overall, the project demonstrates that the open-source GEMS [10] architecture can be turned into a low-cost, modular sensor platform on wheels, providing a reproducible foundation for future research and classroom exercises in embedded communication, control and data acquisition.

Contents

Nomenclature

Abbreviations

Abbreviation	Definition
GEMS	Graceful Equalising of Mechatronics Students
PCB	Printed Circuit Board
CAN	Controller Area Network
BLE	Bluetooth Low Energy
GUI	Graphical user interface
FSM	Finite State Machine
MCU	Micro Controller Unit
TCP	Transmission Control Protocol
VOC	Volatile Organic Compound
SoftAP	Soft Access Point
RH	Relative Humidity
ToF	Time of Flight
UDP	User Diagram protocol
SYN	Synchronize
ACK	Acknowledgment
GPIO	General Purpose Input/Output
I²C	Inter-Integrated Circuit

Symbols

Symbol	Definition	Unit
V	Velocity	[m/s]
ΔS	displacement per time step	[m]
GR	Gear Ratio	-
RPM	revolutions per minute	-
T_s	Sampling interval	[s]
W_R	Wheel Radius	[m]
V_{sound}	Speed of the sound	[m/s]
d	distance	[m]

1 Introduction

1.1. Introduction from Whole Group

This project aims to implement a mobile robot that can collect spatially distributed data inside a building. By driving through corridors and rooms instead of relying on fixed sensor nodes, the platform records how environmental variables, temperature, humidity, air quality indices, and structural stresses change from one location to the next. Such location-aware data helps facility managers spot hot spots, detect potential failure points and optimise control strategies more effectively than a single, static measurement could.

To keep costs low and the system easy to reproduce, the robot is built on the existing GEMS [10]. architecture: four stackable printed-circuit modules (power, sensing, communication and motor control) connected with a CAN bus. Our contribution is twofold. First, we adapt this architecture to a mobile setting by integrating ultrasonic ranging, Wi-Fi telemetry and an on-board state machine for autonomous navigation. Second, we demonstrate that the GEMS stack can serve as a flexible “sensor station on wheels,” ready for further research or teaching in distributed data collection.

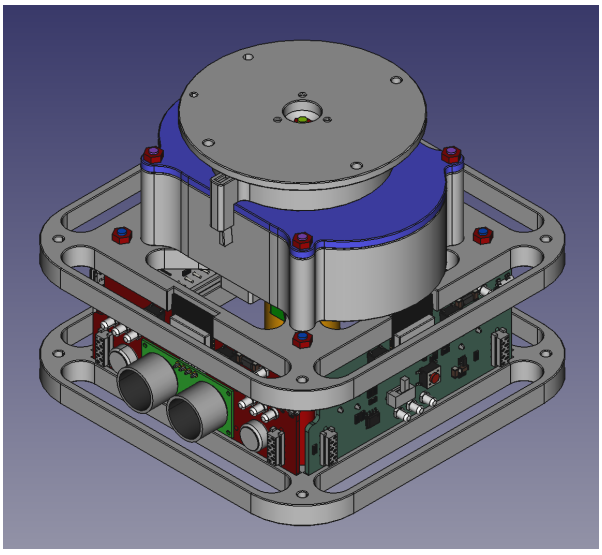


Figure 1.1: GEMS architecture

1.2. State-of-Art Analysis

In modern embedded systems, we often find powerful microcontrollers such like the ESP32 paired with sensors, motors, and displays to form smart, interactive devices. Technologies such as the CAN

bus are widely used in vehicles and industry for robust communication between devices, while Wi-Fi enables remote control and monitoring.

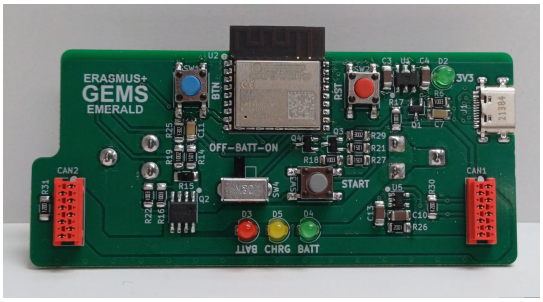
In many existing systems, components like environmental sensors, motor controllers, user interfaces, and power monitors are built as separate, standalone modules. Each part works well on its own but doesn't communicate or cooperate with the others. As a result, building a complete system often requires manually managing multiple boards and writing custom code for each one.

1.3. Problem Definition

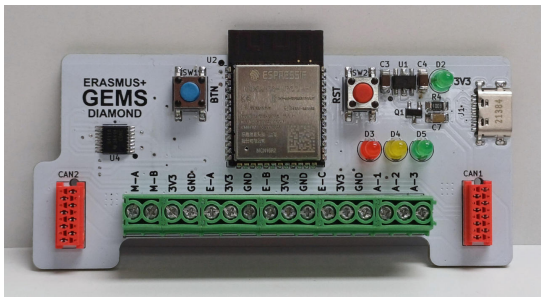
1.3.1. The Situation

Originally, our system had four separate PCBs (Emerald, Diamond, Ruby, and Sapphire), each responsible for a different function:

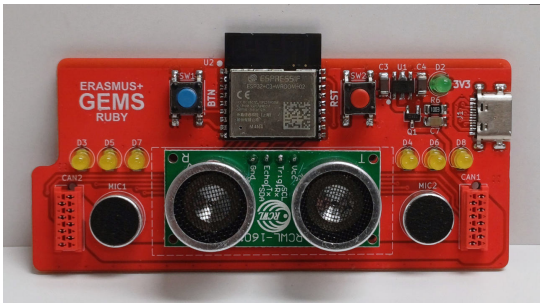
- **Emerald**: Handled power supply, monitoring and system switching.
- **Diamond**: Manage motor control.
- **Ruby**: Provide sensor that measure distance and LED feedback. Transmit data from modules to host PC.
- **Sapphire**: Captured environmental data such as temperature, humidity, and air quality. Provide stable Wi-Fi signal for data transfer.



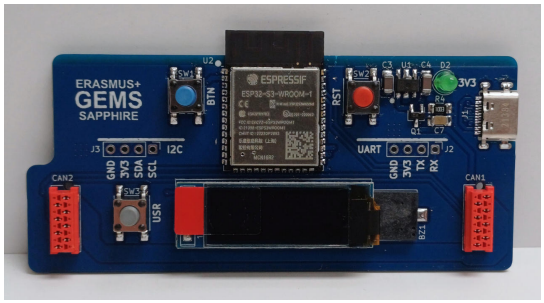
(a) Emerald PCB front view



(b) Diamond PCB front view



(c) Ruby PCB front view



(d) Sapphire PCB front view

Figure 1.2: 4 PCBs used in our project

Each module worked independently, but there was no centralized communication or coordination.

1.3.2. Group Division

To manage the design and integration of all PCBs and systems, the team is divided into three specialized subgroups. Each subgroup is responsible for specific modules and tasks. The Wireless Communication & Sensing Group handled the Sapphire and Ruby modules. The Motor Control Group was in charge with the Diamond module. The power & Mechanical Group manage the Emerald module and the chassis. Our subgroup was responsible for the Wireless Communication & Sensing.

1.3.3. What We Did

We developed a solution that takes all four modules into one integrated system:

- All modules communicate over multiple CAN bus. Where reserve bytes are made for future development.
- The system allow real-time interaction with a web serve on host PC via Wi-Fi.
- Data from sensors is send to host PC and plotted as heatmaps.

This integration enables synchronized control and data sharing, making the system more intelligent, extendable, and easier to use.

1.3.4. Why It Matters

Without integration, systems are harder to manage, cannot adapt in real-time, and waste hardware potential. Manual operations are slow and inaccurate. Our unified setup improves usability, adds new capabilities (like remote data logging and live control), and opens the door for more advanced applications such as environmental mapping and autonomous behaviour.

1.4. Thesis Synopsis

This thesis presents the design and implementation of a unified, real-time embedded system composed of four formerly separate modules. The system:

- Uses CAN bus for internal communication between hardware modules and ESP-now between two robots.
- Connects to host PC via Wi-Fi, transmitting sensor data in real-time.
- Displays sensor readings on an OLED screen and allows users to control LEDs and motors remotely.
- Logs environmental data (temperature, humidity, VOC) to a CSV file via a Python-based server on a computer.

All the above result in a flexible, extensible, and interactive system that combines sensing, actuation, and real-time communication in a unified architecture.

2

Literature Study

In this chapter, we are going to provide the theoretical background for the choice of MCUs and the choice of communication protocols.

MCU ESP32

The ESP32-C3-WROOM-02 is a compact, low-power Wi-Fi and Bluetooth-enabled microcontroller module based on the RISC-V architecture. It is especially well-suited for embedded robotic applications requiring secure, wireless communication with constrained resources. As highlighted in the literature, this module provides a range of critical features that make it an attractive choice for modular and distributed robotic systems. There are also other MCUs available, such as STM32 [6] and Raspberry PI [13]. In Table 2.1, we compared the pros and cons of these three MCUs.

Table 2.1: Comparison of ESP32, STM32, and Raspberry Pi for embedded applications

Aspect	**ESP32 (C3/S3)**	**STM32 (F4/G0 class)**	**Raspberry Pi (Zero 2 / 4)**
Processing core	1–2 × RISC-V or Xtensa @ 160–240 MHz; hardware FPU on S3	ARM Cortex-M (0–4/7) @ 48–180 MHz; optional DSP/FPU	Quad-core ARM Cortex-A53 or A72 @ 1–1.5 GHz; full MMU, Linux OS
Integrated connectivity	2.4 GHz Wi-Fi, BLE, ESP-NOW, plus on-chip CAN (TWAI)	No native Wi-Fi/BLE; CAN on selected parts; ext. radio required	Gigabit Ethernet, USB, HDMI; Wi-Fi + BLE via on-board module
Typical power draw	20–80 mA active; <5 µA deep-sleep	10–40 mA active; <2 µA stop-mode (no radio)	150–700 mA running Linux; ~50 mA idle with Wi-Fi off
Unit cost (USD)	$2–$4 MCU incl. RF	$3–$8 MCU; +$4–$8 for add-on Wi-Fi	$10–$55 board (Zero 2 W → Pi 4), incl. SoC, RAM, storage connectors

The review emphasises the ESP32-C3's affordability, accessibility through Arduino and ESP-IDF frameworks, and wide community adoption. These factors collectively support its use as a communication and control node in modular robotic systems, particularly in sensor hubs and FSM-driven modules such as Sapphire.

2.1. Communication Protocols

Controller Area Network (CAN Bus)

The Controller Area Network (CAN bus) protocol is widely adopted in distributed embedded systems, particularly in automotive and industrial applications. It provides a robust and efficient means of communication among multiple microcontrollers sharing a common communication medium. There are also other protocols like LIN [15] and SENT SAE-J2716 [17]. According to the literature, its relevance to mobile robotic platforms arises from several inherent features:

Table 2.2: Comparison of three automotive serial interfaces

Aspect	CAN 2.0 / ISO 11898	LIN 2.x / ISO 17987	SENT (SAE J2716)
Typical data-rate	20 kbit/s – 1 Mbit/s (up to 5 Mbit/s for CAN-FD)	Fixed 19.2 kbit/s (16× oscillator)	30 bit periods – 90 µs per nibble[1] (≈ 166 kbit/s max)
Bus topology	Multi-master differential two-wire (CAN_H/CAN_L)	Single-master, single-wire with dominant/recessive states	Unidirectional point-to-point (sensor → ECU)
Frame/payload	8 data bytes (64 bit) in Classical CAN; 64 bytes in CAN-FD	2–8 data bytes per response frame	4- or 6-nibble payload + CRC nibble; optional status pulses
Error detection	Bit-monitoring, stuff-bit checks, CRC-15, ACK	1-byte parity + classic checksum (in□frame)	4-bit CRC nibble per message; Manchester-encoded timing check
Synchronisation	Self-clocking, no global time-base required	Synch break + synch byte issued by master	Rising edge of each tick synchronises timing; no clock line
Typical use-case	Power-train, chassis, body, industrial automation	Low-cost body electronics (window lift, seat, HVAC)	High-resolution sensors (throttle, pressure, speed) replacing analog ratiometric signals
Node count	Up to 110+ per segment	16 slaves per master (practical)	One sender, one receiver
Implementation cost	Medium; transceiver + MCU with CAN core	Low; LIN transceiver + low-end MCU/UART	Very low; digital output on sensor, timer input on ECU

In robotic architectures reviewed in Chapter 4, the CAN bus is often used as the backbone for module-to-module communication. Each functional unit (e.g., sensing, actuation, power) is assigned a unique CAN ID, and system-wide communication is coordinated without the need for a central controller. This architecture supports hot-swapping, modular upgrades, and redundancy—key factors in scalable robot design.

Inter-Integrated Circuit (I^2C)

I^2C is a lightweight, synchronous serial communication protocol used extensively in embedded systems for short-range, low-speed data exchange. It is particularly well-suited for connecting sensors to microcontrollers due to its simplicity and low overhead. In embedded systems, UART [16] is also widely used. Chapter 4 identifies the following benefits of using I^2C:

[1] SENT encodes each 4-bit nibble as $12 + X$ clock ticks; with a 3 µs tick, the fastest nibble takes 90 µs, corresponding to roughly 166 kbit/s for a 6-nibble message.

Table 2.3: Key differences between I^2C and UART serial interfaces

Aspect	I^2C (Inter-Integrated Circuit)	UART (Universal Asynchronous Receiver/Transmitter)
Signal lines	2 wires: **SDA** (data) and **SCL** (clock) with pull-ups	2 wires: **TX** and **RX** (plus optional RTS/CTS for flow control)
Clocking	Synchronous; clock supplied by bus master	Asynchronous; each node uses its own baud-rate clock
Topology	Multi-master, multi-slave shared bus	Point-to-point; one TX pin to one RX pin (or via multidrop RS-485)
Addressing	7- or 10-bit device addresses in every transaction	None; link is dedicated to the connected peer(s)
Typical data-rate	Standard 100 kbit/s, Fast 400 kbit/s, Fast+ 1 Mbit/s, Ultra-Fast 5 Mbit/s	Common 9.6 k–921.6 kbit/s; up to 5 Mbit/s on modern MCUs
Hardware cost	Requires two pull-up resistors; open-drain drivers on MCU pins	No extra passives; full-swing push-pull drivers already in MCU
Use cases	Sensor clusters, EEPROMs, ADC/DACs on a short PCB bus	Debug consoles, GPS modules, Bluetooth modems, long-cable links
Pros	One bus can host many devices; clock stretching for slow peripherals	Simple, streaming friendly, long cable runs possible, minimal protocol overhead
Cons	Limited bus length/capacitance; address collisions; shared-bus contention	Only two endpoints unless extra transceiver; fixed baud-rate tolerance required

I^2C is typically employed in local communication between the main control microcontroller and peripheral devices such as ultrasonic sensors, temperature/humidity sensors, and gas detectors. In the systems reviewed, I^2C complements CAN by offloading sensor acquisition from the main control bus. While CAN handles inter-module communication and state coordination, I^2C handles data collection at the node level.

ESP NOW

ESP-NOW is a commonly used wireless communication protocol developed by Espressif for short-range and low latency data transmission. It enables easy and direct peer-to-peer communication between the microcontroller of ESP32 devices without the need of a Wi-Fi connection or any access point [7]. ESP-NOW is typically employed for real-time communication between distributed modules in a mobile robotic system. In Chapter 4, the benefits of using ESP-NOW will be further discussed.

3

Program of Requirements

3.1. Main Program of Requirement

- **General**:

  The robot must operate autonomously for at least 30 minutes, log data in a CSV format wirelessly, and is able to avoid obstacles using onboard sensors.

- **Power**:

  The robot is powered with battery (1.5-6 V), internally regulated to 3.3 V. battery status must be monitored, with auto-shutdown at 0 % SoC.

- **User interface**:

  Start/stop buttons, battery switch, and led indicator for power, changing, and fault status must be accessible for the users.

- **Hardware**:

  Two DC motor-driven wheels, sensing mounting support, modifiable design, a suitable speed ratio (5-1000 RPM), and cost-efficient.

- **Communication & Control**:

  Uses CAN bus for internal communication with inputs: motor duty cycle, turning amount, and sensor distance. Wireless connection to host device required. Obstacle detection and sensor synchronous must be handled.

- **Safety & protection**:

  Overcurrent and overtemperature protection included. ESP protection compliant with IEC 61000-4-2 level 4.

3.2. Subgroup Program of Requirement

3.2.1. Functionality Requirements

Compulsory Requirements

1. The robot must autonomously detect and avoid obstacles using onboard sensors.
2. The system must wirelessly communicate with a host PC.
3. The robot must monitor environmental variables (temperature, humidity, air quality).
4. The robot must log operational and environmental data during runtime.
5. The system must visualize mapping and robot state in real time.
6. The robot must maintain internal synchronization across all modules.

Optional Requirements

7. The system should support a manual override control mode via the host PC.
8. The robot should detect and report sensor faults or lost connections.
9. The mapping functionality should allow recovery from partial data loss.
10. The robot should issue warnings if environmental readings exceed safe levels.

3.2.2. Implementation Requirements

Compulsory Requirements

1. **Modular Communication via CAN Bus**
   - All modules (Power, Drive Control, Sensing, Communication) must support CAN 2.0B at a baud rate of 250 kbps.
   - Each module must use a unique message ID and follow the standard 8-byte CAN frame format.
   - Modules must detect and respond to bus error frames within 10 ms.
   - Message delivery success rate must be $\geq$ 99.5% under nominal conditions.
2. **Wireless Communication with Host PC and Between Modules**
   - Wi-Fi link must maintain a minimum bandwidth of 1 Mbps.
   - Wi-Fi range should be at least 10 m inside the building.
   - ESP-NOW must achieve packet latency of $\leq$ 20 ms.
   - Reconnection must occur within 2 seconds of a connection drop.
   - Packet loss must remain $<$ 1% over a 5-minute test period.
3. **Environmental Sensing Capability**
   - Sensors must measure:
     - Temperature: 0–50°C, accuracy $\pm$0.5°C
     - Relative Humidity: 20–90% RH, accuracy $\pm$3% RH
     - VOC: resolution 1 ppm, update rate $\geq$ 1 Hz
   - Sampling interval must be configurable between 1–5 seconds.
   - All sensor readings must be timestamped with millisecond precision.
4. **Ultrasonic Sensing and Motor Control Logic**
   - Three ultrasonic sensors must be mounted facing front, left, and right.
   - Sensor range: 0–200 cm, with $\pm$1 cm accuracy at 30 cm.
   - FSM must process readings and update motor control within 100 ms.
   - FSM states must include: `forward`, `turn_left`, `turn_right`, `stop`.
5. **Data Storage and Logging**
   - Logged fields: `timestamp`, `X`, `Y`, `temperature`, `RH`, `VOC`.
   - System must log at least 1000 records per session.
   - Data must be transmitted to TCP server at $\geq$ 1 Hz without packet loss.
   - TCP server must store data in real-time with 0% loss over 10 minutes.
6. **2D Grid Mapping**
   - Data must be mapped to (X, Y) coordinates on a 2D grid.
   - Grid resolution must be $\leq$ 0.1 m per cell.
   - Grid must update within 500 ms after new data is received.

7. **Real-Time Data Visualization**
    - GUI must update sensor display at $\geq$ 1 Hz.
    - GUI must display: robot path, FSM state, sensor values, and obstacle markers.
    - User must be able to pause/resume/export data without interrupting logging.
8. **System Synchronization**
    - System clocks must synchronize to within $\leq$ 50 ms drift.
    - Timestamp alignment between CAN and Wi-Fi data must be accurate to $\pm$100 ms.
    - Synchronization must be validated at least once per minute.

4

Design specification

4.1. System architecture overview

The system integrates four functional modules, Power Control, Motor Control, Sensing, and Communication into an embedded platform designed for sensor fusion, environmental monitoring, and actuation. Each module plays a distinct role in the system and communicates over a central bus architecture. A key feature of the system is its ability to wirelessly transmit collected data to a remote host via TCP/IP server.

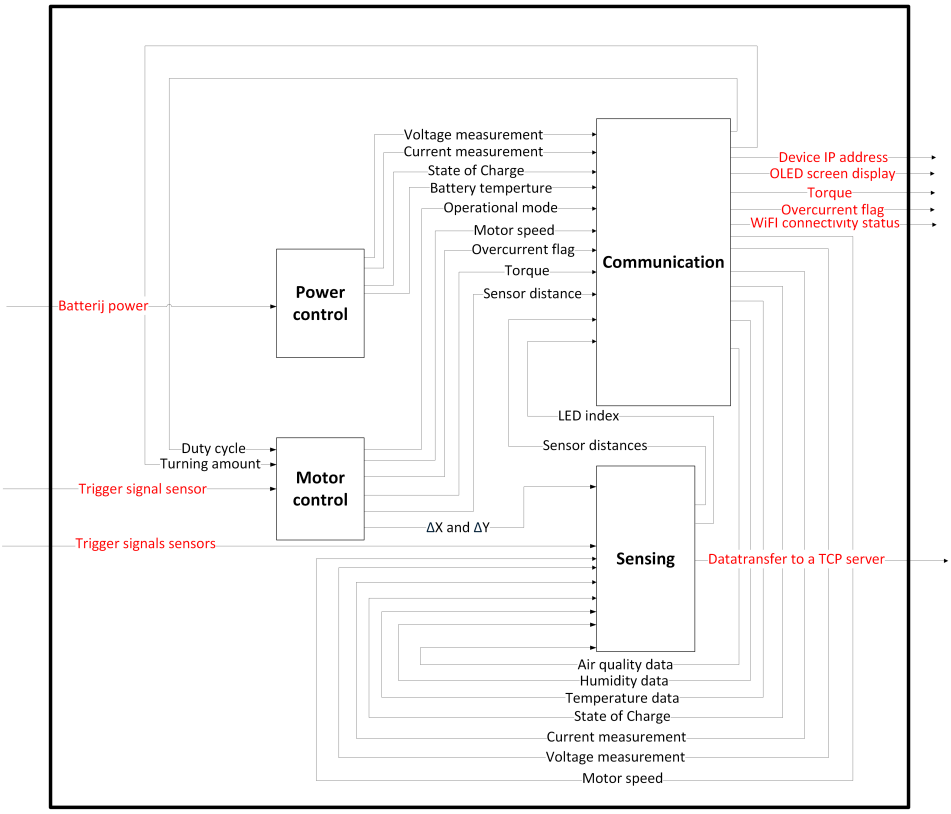


Figure 4.1: I/O Protocol

At the heart of the system's connectivity is the Communication Module, which handles user commands, interface display, and most critically, wireless data transmission. This module connects to a local Wi-Fi network (using Sapphire) as an Access Point and acts as a TCP server, enabling external clients such as a PC or mobile device to connect directly.

The Communication Module receives processed data from all other modules—including temperature, humidity, air quality, battery status, and motor speed and sends these datasets wirelessly to the connected client. The transmission follows a predefined TCP protocol, sending sensor data in real time, which can be used for live monitoring or logging.

4.2. Modular communication via CAN bus

The CAN bus acts as the central communication medium, interconnecting all modules in a multi-master broadcast system. Each module is assigned with an unique CAN ID and transmits or listens for frames based on its role. This allows for asynchronous, prioritized, and non-blocking data exchange, essential for real-time embedded systems.

- **Power Control Module**: Periodically broadcasts voltage, current, state-of-charge, battery temperature, and power readings to the network using unique CAN identifiers. The Communication Module receives these messages for display and further transmission.
- **Motor Control Module**: Listens for control commands (duty cycle, turning amount) from the Communication Module and responds with its operational data, including motor speed, torque, and fault flags. These are published as CAN frames at regular intervals of 10 ms.
- **Sensing Module**: Collects environmental measurements (temperature, humidity, air quality, and distance to obstacles) and sends them via CAN to the Communication Module. It also receives trigger signals over CAN to synchronize data collection or LED indicators.
- **Communication Module**: Serves as the CAN bridge and gateway to external systems. It listens to all relevant CAN frames, aggregates the data, and forwards it to the Wi-Fi interface. It also generates control frames (e.g., motor commands, LED control) and sends them via CAN to other modules.

4.3. Wireless communication design

The system architecture is modular and distributed, with each functional block—Power Control, Motor Control, Sensing, and Communication—interconnected via a CAN bus for wired inter-module messaging. However, wireless communication becomes essential when transferring data to external devices or users.

The Communication Module acts as a wireless interface, aggregating data from all other modules over CAN and transmitting it wirelessly. This allows real-time monitoring and control without the need for manual operation of the system. Specifically:

- Sensor data (temperature, humidity, air quality, distance to the obstacles)
- Power metrics (voltage, current, SoC)
- Motor status (speed, duty cycle, torque)

All are collected and displayed wirelessly.

4.3.1. Wi-Fi communication with host PC

To support external data logging and remote interaction, the system sets up a Wi-Fi Access Point (AP) [21] directly from the ESP32-based Communication Module. This allows a host PC or mobile device to connect directly to the module over Wi-Fi without requiring an external router.

Once connected, a TCP socket server runs on the host PC, while the ESP32 functions as a TCP client that sends sensor data periodically (every seconds) in a comma-separated format: [timestamps, x, y, temperature, humidity, air quality]. This is used to stream real-time environmental and sensor data to the PC, then visualize the data and log it into a CSV file for later usage.

The design also supports bidirectional extensions, where the PC could send commands back to the ESP32 over the same TCP channel, enabling interactive control features such as mode switching or data request on demand.

4.3.2. ESP-NOW communication between modules

To enable wireless communication between two modules in our embedded system, which itself uses two ESP32 chips, each equipped with an antenna. These modules are designed to be able to exchange data wirelessly in both directions. For example, the left communication module might send commands like motor control or LED activation to the right module. At the same time, the right module can also send back the data to the left module such as sensor feedback or status updates. This setup requires a reliable, low-latency, bidirectional communication link, which is why ESP-NOW is used.

ESP-NOW is a communication protocol developed by Espressif, specifically designed for ESP32 chip and other ESP chips[7]. This communication protocol allows multiple devices to exchange small packets of data without needing a Wi-Fi connection or router, instead of using widely used TCP/IP, ESP-NOW uses a connectionless protocol based on Wi-Fi's MAC addresses, which will eventually significantly reduces overhead and latency.

In terms of performance:

- Transmission time for one packet is typically around 1-3 milliseconds, making it suitable for real-time or near real-time applications.
- Each ESP-NOW packet can carry up to 250 bytes of payload data, including any application-specific information (e.g., control commands, sensor values).
- The protocol supports sending data to up to 20 peers simultaneously, making it easier to carry out future expansion.

In our project, ESP-NOW is an ideal solution because:

- It supports peer-to-peer communication, which fits our need for two-way data exchange.
- It allows devices to communicate without having to connect to an existing Wi-Fi network.
- It offers low power consumption, which is beneficial for embedded system like ours.
- It supports both broadcast and unicast messaging, allowing flexibility in addressing devices.

4.4. Sensor Integration and environmental monitoring

To measure indoor environmental conditions, two digital sensors were selected: the Sensirion SHT40 for temperature and relative humidity (RH) measurement, and the Sensirion SPG40 for detecting VOCs. These sensors were chosen after comparing multiple available options based on their accuracy, interface compatibility, and simply the cost.

4.4.1. Sensor selection

From the sensor list in Table 4.1, our group prioritized devices that met the following criteria:

- High accuracy, with a temperature accuracy ≤ ±0.3°C and humidity ≤3% RH
- I^2C digital communication interface, compatible with the ESP32 micro-controller
- Low cost, to keep the overall system affordable for the clients

Table 4.1: Comparison of temperature and RH sensors options

Sensor	Temp. accuracy	Humidity accuracy	Interface	Supply voltage	Price (€)
SHT31	±0.3°C	±2% RH	I^2C	3V / 5V	13.37
SHT31-F	±0.2°C	±2% RH	I^2C	3V / 5V	12.23
SHT35	±0.1°C	±1.5% RH	I^2C	3V / 5V	14.80
SHT40	**±0.2°C**	**±1.8% RH**	I^2C	3V / 5V	**7.03**
AHT20	±3°C	±2% RH	I^2C	2V–5.5V	8.12

The SHT40 was selected for its compact design, high accuracy and fine resolution (0.01 °C and 0.01 % RH). Its I^2C interface [19] allows integration with the communication module, which has free I^2C channels available. Compared to alternatives like SHT31, or SHT35, the SHT40 provides a strong balance

between performance and price.

In addition to temperature and humidity, indoor air quality was measured by selecting a sensor capable of detecting volatile organic compounds (VOC). There are more specialized sensors exist on the markets for measuring specific air components such as CO_2, NO_2, or other particulate matters. VOCs represent a broad category of harmful gases in the air. Measurement of VOCs provides a generalized and practical indicator of indoor air quality.

The decision to choose VOCs was guided by their simplicity, relevance, and easy integration. VOC data are easier to interpret in the context of general indoor air quality and require less supporting components. The SGP40 sensor was selected because its output has a normalized VOC index and is designed specifically for indoor environments. This device also supports I^2C communication, making it suitable for the architecture of the system. In addition, the power consumption is very low and it contains an integrated temperature and humidity compensation making it a good choice. Both sensors are shown in Figure 4.2 and Figure 4.3.

Figure 4.2: Temperature&Humidity sensor SHT40 [5]

Figure 4.3: Air quality sensor SGP40 [4]

4.4.2. Sensor placement

There are four possible mounting positions for the two sensors on the 3D-printed components: front, back, top and bottom, as illustrated in Figure 4.4. The front area is already occupied by the three ultrasonic sensors [1], indicates by the red arrow, and therefore cannot be used for additional components.

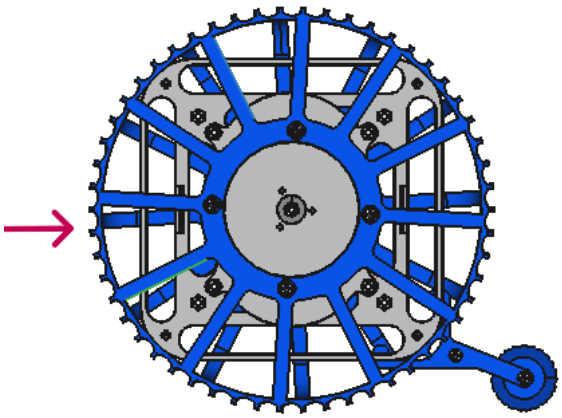


Figure 4.4: Side view of the robot, where support robot is at the back side

Mounting the sensors on the top side seems like a viable option, but this location presents some im-

portant drawbacks. Firstly, when the robot operates near a window, direct sunlight may shine on the SHT40 sensor, potentially compromising the accuracy of its temperature and humidity readings. Secondly, airflow generated by the robot's movements could further influence the sensor's performance.

Placing the sensors on the underside of the robot seems also not ideal. This position increases the likelihood of dust accumulation, which will degrade sensor performance in the long time, and the risk of physical damage will raise due to proximity to the ground.

The remaining option is the backside of the robot. This location avoids the issues associated with the other mounting positions, offering protection from the direct sunlight, physical contamination and airflow interference. For these reasons, the rear of the robot is considered the most suitable position for mounting the sensors in this design.

4.4.3. Sampling strategy

Our group set the sample rate at 1 Hz (once per second) for both sensors. Since temperature, humidity, and air quality typically change slowly over time, collecting data every second is frequent enough to get meaningful environmental variations without loading with excessive data to the system.

By using this sampling rate, the power consumption will also be reduced and processing demands on the microcontroller becomes more efficient. Because our system includes wireless transmission and real-time plotting, limiting data volume is important to maintain nice performance and reliable communication.

4.5. Ultrasonic sensing and motor control

To realise effective obstacle avoidance, a robot must be capable of perceiving its real-time surroundings. There are several advanced technologies available to achieve this functionality. For instance, radar[20] systems can be employed to generate comprehensive environmental maps, while LiDAR sensors offer high-resolution detection of objects by using laser-based distance measurements [11]. However, these solutions often come with considerable cost and require a high level of technical expertise for integration and operation.

Given the constraints of our project, specifically limited financial budgets and technical proficiency. The team have opted for a more accessible and cost-effective solution: the use of ultrasonic distance measurement sensors. These sensors (RCWL-1601) [18] operate by emitting high-frequency sound waves and measuring the time it takes for the echoes to return after bouncing off nearby objects. This time-of-flight (ToF) data allows the system to calculate the distance between the sensor and any object directly in front of it.

Table 4.2: Finite-State Machine

Sensor L	Sensor R	Sensor F	State	Description
0	0	0	Stop	Wall
0	0	1	left_right	Obstacle on left and right
0	1	0	right	Obstacle on left and front
0	1	1	right	Obstacle on left
1	0	0	left	Obstacle on right and front
1	0	1	left	Obstacle on right
1	1	0	left_right	Obstacle on front
1	1	1	forward	No obstacle

When an object is detected within a predefined range, the sensor triggers a response by sending the distance data to the Sapphire. This module houses a finite-state machine (FSM), as shown in Table 4.2, where ”0” indicates an object detected and ”1” indicates no object detected. FSM is responsible for interpreting incoming data and determining the appropriate response. Based on the current state and the detected distance, the FSM processes the information and outputs a directional command, such as turning left, turning right, or stopping, to guide the robot around the obstacle. This approach allows

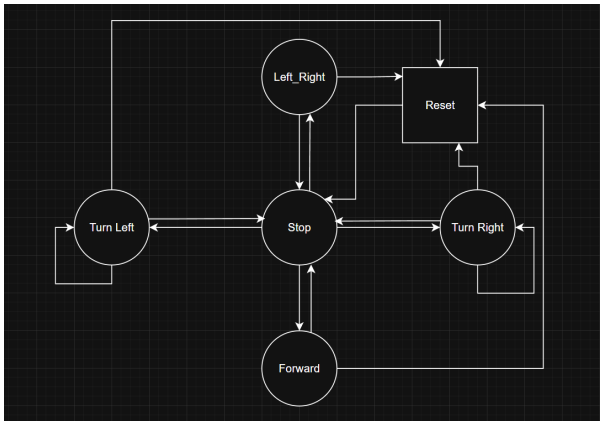


Figure 4.5: FSM Diagram

for real-time, automated navigation adjustments using minimal hardware and computational resources, making it both practical and efficient for budget-constrained projects. In Figure 4.5, the FSM diagram is given. All the states operate first through the stop state before going to the other states. Each time the system resets, it all goes to the stop state first. In case of turning 180 degrees, it operates as turning twice.

4.5.1. Design choice

The initial design of the Ruby, incorporates one ultrasonic distance sensor and two microphones capable of detecting acoustic echoes [19], shown in Figure 4.6. The microphones theoretically provide the capability for acoustic localization by analyzing the time delay between signals received from reflecting surfaces. This technique can help identify the position and movement of sound-reflecting objects within the environment.

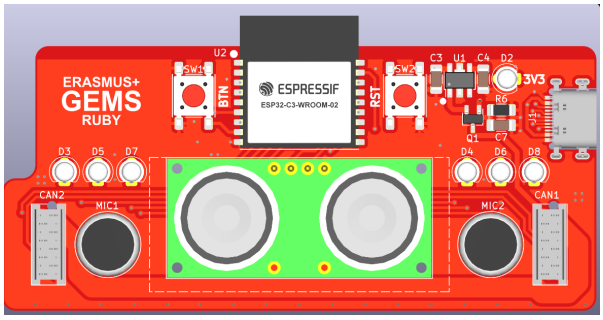


Figure 4.6: Ruby Module

However, in practice, this approach presents several challenges. One major limitation is environmental white noise[14], which can significantly degrade the quality and reliability of echo detection. Differentiating meaningful echoes from ambient acoustic interference becomes highly complex, especially in uncontrolled or noisy environments. Additionally, the effectiveness of this localization technique depends on the presence of sound-emitting or echo-reflective events. In our application, the target objects are passive—they do not emit sound or generate strong acoustic signatures on their own, making it difficult to rely solely on microphones for consistent object detection.

While microphones are still valuable for detecting loud or sudden sounds (such as claps or knocks), they are insufficient for the precision and reliability required for autonomous obstacle detection and avoidance. For this reason, an ultrasonic distance sensor was added to the Ruby module. Ultrasonic sensors actively emit sound waves around 40 kHz [14] and measure the time the echo returns after hitting an object. This active sensing capability ensures consistent detection of objects, regardless of whether those objects emit any sound.

4.5.2. Ultrasonic sensor configuration

The obstacle avoidance system in our robot is built around three strategically positioned ultrasonic sensors: one facing forward, one on the left side, and one on the right side. Each sensor continuously monitors its respective direction and is calibrated to detect objects within a specific threshold distance, ensuring timely responses to potential collisions. The 3D-printed bumper for the sensors are designed together with the hardware subgroup, it is shown in Figure 4.7.

These ultrasonic sensors can detect objects as small as 10 cm by 10 cm within their sensing range. They operate by emitting ultrasonic pulses and measuring the time it takes for the sound waves to reflect off nearby objects and return to the sensor. This method allows the system to accurately calculate the distance to obstacles in real time.

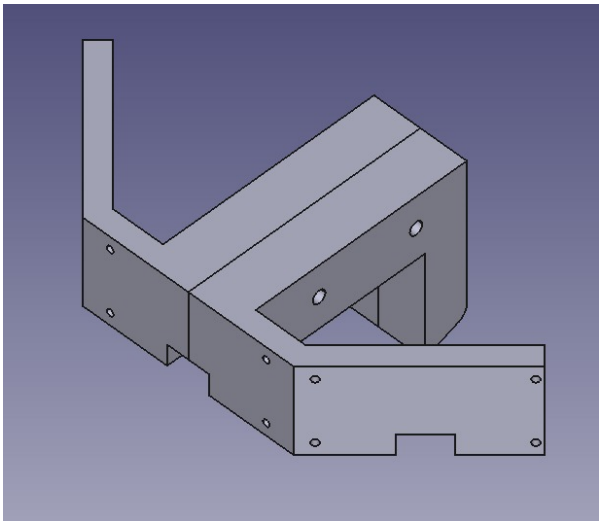


Figure 4.7: 3D Modeling of Bumper

When any of the sensors detects an object that meets the predefined criteria (such as size and proximity), the sensor triggers a signal that is sent to the communication module. Embedded within this module is a FSM shown in Figure 4.5, which serves as the control logic unit of the system.

4.5.3. Threshold Distance and Object Size Justification

The first idea is to make the threshold distance 15 cm, but there are several aspects that changed. And eventually, we have chosen 30 cm as the threshold distance as an optimal sensing distance based on both the geometry of the beam of the ultrasonic sensor and the need for reliable early obstacle detection. The reasons are as shown below:

- **Extended Reaction Time**: At 30 cm, the robot has twice the distance compared to the original threshold to detect and react to obstacles. This is particularly beneficial for maintaining safe navigation at higher speeds or in environments where precise maneuvering is critical.
- **Beam Geometry Advantage**: The ultrasonic sensor has a 15° total beam angle, which translates to an approximate beamwidth of 8 cm at 30 cm distance:

  $$Beamwidth = 2 * (30\ cm * \tan(7.5°)) = 8\ cm \tag{4.1}$$

  This wider beam footprint increases the likelihood of detecting small or partially visible obstacles earlier, reducing the chance of unexpected collisions.
- **Avoiding Near-Field Signal Instability**: Ultrasonic sensors exhibit reduced accuracy at very close range due to signal overlap or dead zones. By setting the threshold at 30 cm, we can avoid a large part of near-field distortion ensuring stable and interpretable echo signals, shown in Figure 4.8.
- **Environmental Noise Rejection**: A longer detection range allows the system to better discriminate between real object echoes and transient background noise, as the signal processing algorithm has more time and data to verify object presence.

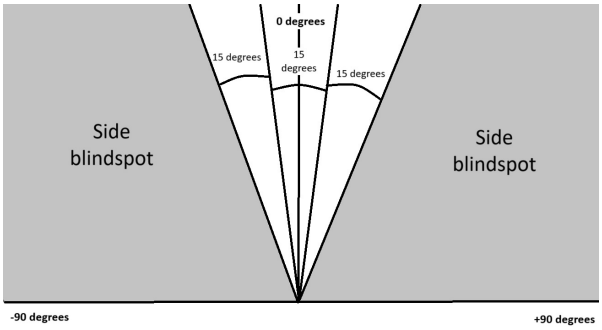


Figure 4.8: Blind Part Demonstration

The selection of 10 cm × 10 cm as the minimum target object size is motivated by the need to minimize dead angles and ensure robust detection within the sensor's beam cone. The reasons are as shown below.

- **Dead Angle Minimization:**
  Dead angles occur when small or thin objects fall outside the beam or produce weak reflections that the sensor cannot detect. A 10 cm-wide object ensures full beam coverage at 30 cm distance, especially considering the 8 cm beamwidth from Equation 4.1, reducing the risk of objects slipping through undetected.
- **Reliable Reflection Surface:**
  With an object height and width of 10 cm, there is sufficient surface area to reflect ultrasonic waves back to the sensor, improving echo strength and consistency. This is particularly important for non-acoustically reflective surfaces (e.g., cloth or rubber), where larger area improves detection reliability.
- **Compact Yet Detectable:**
  The 10 cm $\times$ 10 cm size balances compact form factor with effective detection capability. It allows testing and navigation around realistic indoor obstacles like chair legs, small boxes, or handheld devices—objects common in typical operating environments.
- **Alignment with Sensor Size and Placement:**
  The ultrasonic sensor module on the Ruby board has a physical length of 4.5 cm, centrally positioned between microphones and structural components. After the final integration, the sensors will be taken out of Ruby and will be placed on 3D printed bumper. With a 10 cm-wide object, the echo will reliably return even when the object is slightly off-center, eliminating blind spots caused by narrow beam reflections.

4.5.4. FSM-based motor control

The FSM is the core of the robot's decision-making system, interpreting the real-time input of three ultrasonic sensors positioned to the left, right and front of the robot. These sensors provide binary feedback, with ”1” indicating that there is no obstacle and ”0” indicating an obstacle detected within a predefined threshold. Using this binary input, the FSM identifies the spatial context of any detected obstacles and transitions into the appropriate control state.

As shown Table 4.2, the FSM uses sensor data to determine the safest direction to proceed. For example:

- If all three sensors return ”0”, the robot is surrounded by obstacles, interpreted as a wall, and enters the left-right state, potentially backing up or rotating to find a clear path.
- If only the front sensor detects an obstacle (Sensor F = 0), and both side sensors are clear, the FSM still enters the left-right state, recognizing that a forward move would result in a collision.
- When only the left or right sensor detects an object, the FSM responds by steering the robot in the opposite direction.

- If no obstacle is detected by any sensor (1, 1, 1), the FSM enters the forward state and allows the robot to continue moving ahead.

To handle situations where obstacles are detected on both sides or directly in front of the robot, classified as the left-right state, a simple but effective randomization strategy is employed. In such cases, the robot uses a random number generator to determine the turning direction. Specifically, every 0.5 seconds, a floating-point number between 0 and 1 is generated. If the generated number is greater than 0.5, the robot performs a left turn; otherwise, it turns right. This approach introduces nondeterministic behavior that helps the robot escape from tight spaces or symmetrical obstacle configurations where deterministic logic might fail.

Once a state is selected, the FSM sends the corresponding command to the motor driver, which adjusts the robot's motion, whether to stop, turn, or continue forward. This FSM-based navigation logic ensures that the robot adapts intelligently and efficiently to its environment, even with minimal hardware, making it highly suitable for low-cost autonomous systems.

4.6. 2D grid mapping

To enable environmental awareness and navigation logging, our robot implements a lightweight 2D mapping strategy that integrates motion estimation, obstacle sensing, and wireless communication. This method is designed to be efficient and compatible with resource-constrained embedded systems.

4.6.1. Constant Motion Speed

The robot navigates at a provided, fixed, steady speed, which is continuously reported by the motor drive module. This simplifies position estimation, allowing displacement to be calculated using the formula:

$$\text{Distance} = \int_{t_0}^{t_1} v(t)\,dt \tag{4.2}$$

where v(t) is the velocity as a function of time, the assumption of constant velocity eliminates the need for real-time accelerometer or wheel encoder integration.

4.6.2. Manual Initialization

At the start of operation, the host system manually defines the robot's initial position and orientation. This serves as the origin in the coordinate system and forms the basis for all subsequent position estimates.

4.6.3. Turn Detection and Directional Updates

The robot includes a finite-state machine (FSM) that governs obstacle avoidance behaviour. Whenever the FSM detects an obstacle and initiates a directional change (e.g., turn left or right), a turning signal is transmitted wirelessly to a host PC over Wi-Fi. These turning events allow the system to update the robot's heading and ensure that the calculated trajectory accurately reflects its path through the environment.

4.6.4. Stepwise Environmental Sensing

During each movement step, the robot collects environmental data using its onboard ultrasonic sensor. At regular intervals (based on time or displacement), the system logs:

- The robot's estimated position
- Direction
- Obstacle presence and relative location

This data is formatted and prepared for transmission to the host PC.

4.6.5. Grid resolution

To make sure that our spatial mapping are efficient and the localization is accurate, it is essential to define an appropriate resolution for the 2D grid that represents the robot's environment. This grid must balance spatial precision with computational efficiency to the given constraints of this embedded system used on the robot.

The robot operates at a constant speed and updates its position at fixed time intervals. The displacement per time step ΔS, is calculated using the following equation:

$$\Delta S = RPM * GR * T_s * 2\pi R_w \tag{4.3}$$

Where RPM is the motor revolutions per minute and is set to a constant value of 1000 by the drive group, GR is the gear ratio (motor-to-wheel reduction), which is $\frac{1}{200}$. Ts is the sampling interval, equals to 220 ms. R_w is the wheel radius, which is $8.5\ cm$.

substitute this values gives:

$$\Delta S = \frac{1000}{200} * \frac{220}{1000 * 60} * 2\pi * 0.085 \approx 0.0098\ m \tag{4.4}$$

This means that in every sampling interval the displacement is $0.0098m$. So in one second the displacement is:

$$\Delta S = \frac{1}{0.220} * 0.0098 \approx 0.045\ m = 4.5\ cm \tag{4.5}$$

This result indicates that the robot travels approximately $4.5\ cm$ between the position update in one second. To reflect this physical motion accurately on our 2D map while also avoiding unnecessary computational overhead, our team have selected a grid cell resolution of 5 cm.This resolution ensures that each cell approximately equals a second displacement, simplifying our localization and mapping logic.

4.6.6. Wireless Communication and Host-Side Processing

All collected data and control signals (such as turning events) are transmitted to the host PC via Wi-Fi. A custom Python script receives and processes the data in real-time on the host PC. The script performs the following tasks:

- **Path Visualization**: The robot's movement path is plotted live, showing the current trajectory and locations of detected obstacles.
- **Data Logging**: Each step's data is stored in a structured format and written to a .CSV file. This file includes time stamps, X and Y positions, orientations, and sensor readings, allowing for offline analysis and future playback.

The map can be visualized dynamically or reconstructed post-run using the logged CSV data. This results in a clear, interpretable view of the environment as perceived by the robot.

4.7. Data storage and transmission

During the initial stages of system development, the ESP32 was employed for both data transmission and local data storage. However, limitations in the ESP32's onboard memory restrict the amount of data that can be stored. According to the datasheet [8], ESP32-C3 features approximately 400 KB of SRAM, which needs to be shared between system tasks, buffers, and program execution. In addition to that, the default flash memory available is 4 MB, which may be enough for light data logging, but is not ideal for storing large volumes of time-series sensor data over long extended periods.

Due to these limitations, attempts to store large datasets locally on the ESP32 led to performance issues such as memory overflows and data loss. To solve this problem, it was decided to offload the data via a wireless connection to the host PC. This implementation is based on TCP communication

between the ESP32 and a host device, in this way, the data that has to be transmitted are packed in smaller and manageable batches, this will significantly reduce the load of the chip's internal memory.

4.7.1. TCP server implementation

To ensure continuous and reliable transmission of data from the ESP32 to the host computer, a TCP server-client model was established. The TCP is one of the core protocol of the internet. It allows two devices to communicate in a reliable and organized way, by making sure that all data sent from one device reaches the other devices without error and also in the correct order. Unlike simple protocol like UDP(User Datagram Protocol), TCP uses a so called "3-way handshake" process that takes place in the TCP establishing and closing connection between two devices. In Figure 4.9 a basic 3-way handshake process is illustrated.

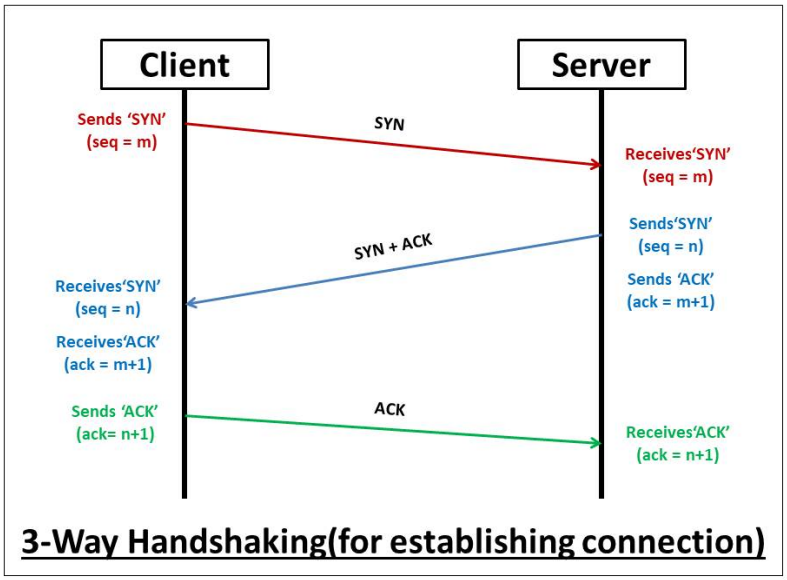


Figure 4.9: 3 way handshake connection [3]

As the name suggests, the three-way handshake involves three steps [9]:

- **Step 1 - SYN (Synchronize**: the client initiates the connection by sending a TCP segment with the SYN flag set. This gives an indication that the clients wants to start communication and includes an initial sequence number to begin the exchange.
- **Step 2 - SYN-ACK (Synchronize-Acknowledgment)**: The server receives the SYN requests ad responds back with a segment that contains both the SYN and ACK flags set. The ACK confirms that the server has successfully received the client's SYN, and a new SYN from the server includes the server's own initial sequence number.
- **ACK(Acknowledgment)**: The client responds then with a final ACK segment, confirming that the SYN-ACK is received. At this point, both the client and server have acknowledged each other's sequence number and a reliable connection is established.

In our system it consists of the following components:

- **ESP32-A (Wi-Fi server)**: Configured to operate in SoftAP mode, this device cerates a local wireless network to which the other ESP32 and the host computer can connect
- **ESP32-B (Data transmitter)**: Configured as a TCP client, this device is able to connect to the SoftAP and transmits all the logged data over a TCP connection.
- **Host PC**: The host PC is connected with ESP32-A's SoftAP Wi-Fi and receives all transmitted data from ESP32-B and logs the data locally in a CSV file.

In this system before the 3-way handshake the server and the client needs to know where the signals have to be send, it's mainly established using IP addresses and port numbers, which are essential for devices identifying. When ESP32-A operate in SoftAP mode, it will creates its own Wi-Fi network and acts as a local access point. By default, ESP32-A uses the IP address 192.168.3.x which acts as the gateway IP address for any device that wants to connect. By setting the ESP32-B to the IP address 192.168.3.x the client can find the server device. But to make sure that the device is really the one that was needed, port numbers can be set for both ESP32-A and ESP32-B. A port number is used alongside the IP address to uniquely identify specific device exactly. In our design it is set as ”12345”, just for simplicity.

5

Prototype implementation and results

In this chapter, our team present a comprehensive, step-by-step account of the implementation process. We have successfully developed fully functional and autonomous modules by meticulously integrating both software and hardware components and incorporating additional external sensors. This implementation demonstrates our modular design's effectiveness and highlights the robustness of the communication architecture that underpins the system's overall performance.

5.1. Communication Module Implementation

The implementation of the Sapphire will be explained. Sapphire is the brain of our robot. All the information comes to Sapphire via CAN. In Figure 5.1 a simple diagram is shown for all the modules are connected.

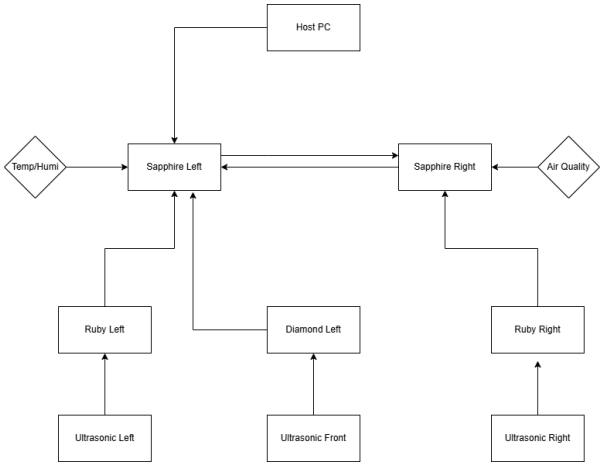


Figure 5.1: Global Communication Transmission Diagram

5.1.1. Inter-Sapphire Communication via CAN Bus

Initially, communication between the left and right Sapphire PCBs was implemented using ESP-NOW, a lightweight wireless protocol. However, significant synchronization issues were observed during testing, particularly under high-frequency data exchange and multi-node coordination. These limitations led to the transition to a wired CAN bus solution.

ESP-NOW test between two Sapphires

In one of our tests by using ESP-NOW, the left Sapphire is set to send a data packet every 2 seconds. The right Sapphire did receive the data, but when it tried to send a response back at the same time, it

caused a problem: the incoming data got overwritten or was not fully received. This means that both sides were talking over each other.

To try and fix this, our team added a receive flag. This is added to make sure that the a device would only send data if it had finished receiving the previous message. This would helping with this talking over issue, but the problem did not go away. Even with the flag, data was still getting lost or corrupted during two-way communication, as shown in Figure 5.2 and Figure 5.3.

```
Received VOC-index: 89.Temp: 26.95
Humiditiy: 47.11
Data sent
Received V.00
Temp: 26.94
Humiditiy: 47.11
Data sent
Data received!
Received VOC-index: 90.00
Temp: 26.94
Humiditiy: 47.06
Data sent
Data received!
Received VOC-index: 90.Temp: 26.94
Humiditiy: 47.09
Data sent
```

Figure 5.2: Left sapphire module sending temp and RH

```
Received temp: 26.95
Received hum: 47.11
voc: 89.00
Data sent
Receeceived hum: 47.voc: 90.00
Data sent
Received temp:
Received hum: 47.06
voc: 90.00
Data sent
Received temp: 26.94
Received hum: 47.09
voc: 90.00
Data sent
```

Figure 5.3: Right sapphire module sending VOC

The left Sapphire module first sent temperature and humidity data to the right Sapphire, and this initial transmission was successfully received. However, when the right Sapphire started sending VOC values back to the left, the data wasn't received correctly. Instead of getting a message like "Received VOC-index: 89.00", the left Sapphire only got "Received V.00", indicating that part of the data was lost during transmission.

After this, when the first Sapphire sent the second set of data to the right, the received message was then "Receeceived hum: 47." instead of the correct temperature value of 26.94 and humidity value of 47.11. These errors showed that the wireless communication was unreliable and the team decided to switch to a system that is fully based on CAN bus communication, which offers better stability and reliability.

CAN Connection inside Module

The CAN bus provides deterministic, real-time communication with built-in error detection and message prioritisation. This change significantly improved the reliability and timing consistency of inter-module communication. Each Sapphire board operates as a CAN node, exchanging structured messages that include sensor data, control flags, and synchronisation signals.

During the coding, a library was used. ESP32-TWAI-CAN[12] is a very powerful library for CAN communication. The sketch configures the ESP32-S3's on-chip TWAI (CAN 2.0 B) controller on GPIO 5 (TX) and GPIO 4 (RX) at 500 kbit/s. Once every certain time, it constructs an 8-byte CAN frame whose identifier field is set to a placeholder value 0xXXX—in automotive practice, a unique 11-bit identifier distinguishes each message type (ID and data frame), so every data stream on the bus can be recognised and filtered purely by that ID. The payload layout is then filled: byte 0 declares the number of useful data bytes, byte 1 indicates the OBD-II service, byte 2 holds the parameter to request, and the remaining bytes are padded with 0xAA to avoid long runs of identical bits that would provoke CAN bit-stuffing. The wrapper call ESP32Can.writeFrame(frame) queues the packet for transmission; the TWAI hardware arbitrates on the shared bus and retries automatically if another node with a higher-priority identifier wins. Incoming traffic is polled with ESP32Can.readFrame(rxFrame, 1000). Whenever a response arrives, the code inspects the rxFrame.identifier; if that value matches the expected unique ID for the reply, the payload is decoded (e.g., converting byte 3 to temperature by subtracting 40 °C) and printed. Thus, every transmitted or received message is tagged by its own unique 0xXXX identifier,

enabling orderly, collision-free communication among multiple devices on the CAN network. Below in Table 5.1, the CAN ID setup is shown.

Table 5.1: CAN-frame map: left- and right-hand module stacks

Left-hand stack				Right-hand stack		
CAN frame	**Description**	**Dir.**	**ID**	**CAN frame**	**Description**	**ID**
voltageLeft		Rec	0x100	voltageRight		0x405
currentLeft		Rec	0x100	currentRight		0x405
ledIndexLeft		Trans	0x200	ledIndexRight		0x500
socLeft		Rec	0x110	speedRight		0x705
distanceLeft		Rec	0x205	dutyCycleRight	From left Sapphire to local Diamond	0x700
distanceFront		Rec	0x300	voc		0x600
speedLeft		Rec	0x305	TempEmeraldRight	Battery temperature	0x415
dutyCycleLeft	From PI	Rec	0x310	distanceRight		0x505
controlState	Controlled by Wi-Fi and FSM	Trans	0x315	socRight		0x405
TempEmeraldLeft		Rec	0x115	checkstate		0x710
Environmental Temp		Trans	0x320	Sendstate		0x720
Environmental Humidity		Trans	0x320	dutyCycleRight_send		0x730

Every CAN message gets its own unique number, so two messages never crash into each other on the network. As shown in Figure 5.4 and **??**, the duty_cycle has been sent to Sapphire. And the duty_cycle has been printed in Sapphire. That lets the hardware automatically give urgent messages first place and ignore anything it doesn't need, which means the main processor has less work to do. We also leave empty numbers in between on purpose: if we ever want to add new or more detailed data later (like individual-cell voltages or more precise air readings), we can just use those spare numbers without having to rewrite the whole system.

```
15:38:18.077 -> Sent duty cycle to Sapphire: 0.40
15:38:18.110 -> 0.40
15:38:20.101 -> Sent duty cycle to Sapphire: 0.40
15:38:20.101 -> 0.40
15:38:22.123 -> Sent duty cycle to Sapphire: 0.40
15:38:22.123 -> 0.40
```

Figure 5.4: Sending and Receiving Duty Cycle From Sapphire

Each Sapphire module interfaces with multiple sensors and PCBs using a combination of I2C and CAN protocols:

- Environmental Sensors: The temperature/humidity and air quality sensors are connected directly to the Sapphire boards via I2C. These sensors provide real-time environmental data, which is processed locally and shared with other modules via CAN.
- Ultrasonic Sensors: Ultrasonic sensors are connected to the Ruby PCBs using I2C. Ruby handles initial data acquisition and formatting, then transmits the processed data to the left Sapphire board over the CAN bus. This layered approach offloads sensor-specific processing from Sapphire and ensures modularity.

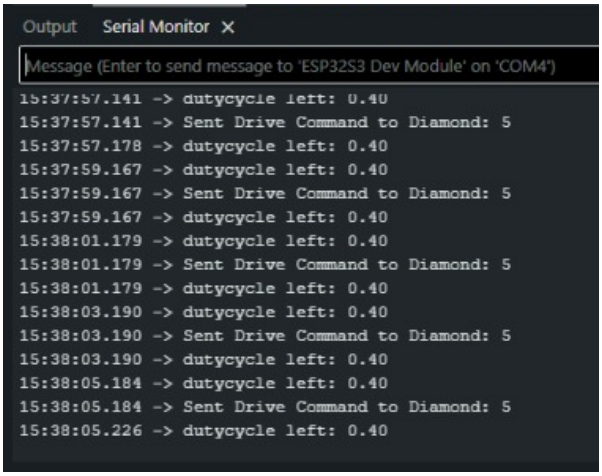


Figure 5.5: Sending Command and Receiving Desired Duty Cycle

5.2. Software Implementation

5.2.1. Web Server

We have made a sketch that turns the ESP32 into a tiny web-server interface for the robot. It has the following functions:

Table 5.2: Webserver Utility

Function	Command
Motor	Stop, Forward(Start)
Shut Down	Shut down the entire system

Except for the commands, on the browser website, all the important variables are shown live in Figure 5.6.

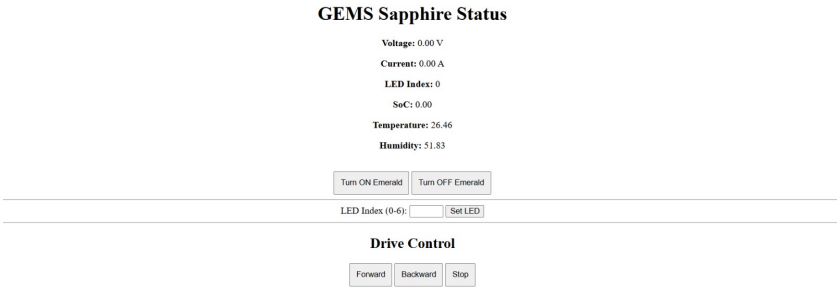


Figure 5.6: Example of Webserver Interface

The layout is minimal; adding new variables is as easy as including another line in the HTML block and appending the matching telemetry print. So, additional data can be dropped later with only a few extra lines of code.

5.2.2. Python Script

At the core of the system there is a python-based TCP socket server. This server listens on all the network interface by using (0.0.0.0) at port 12345, and allowed incoming connections from WiFi-enabled devices, in our case the Arduino on the client device (ESP32). this client operates in Station (STA) mode

and connects to the same local Wi-Fi network as the host computer. The server utilizes the socket module for handling TCP communication, and use threading for concurrently managing real-time updates and user input.

After the script get started, the socket server will first try to bind and begins listening for incoming connections. Once a client connects, it begins receiving transmitted data from the client in the form of newline-bounded strings. Each message starts with a tag to identify its type. Either ”T” for temperature, humidity and air quality, or ”P” for SoC, voltage, current, onboard temperature.

The server parses each message and performs conditional processing based on the tag. For ”T” messages, the script extracts the X-Y coordinates and the corresponding sensor values, and updates the internal data grid. In the meanwhile, these values are also written to a CSV file for permanent storage. For ”P” messages, power data is logged into a separate CSV file, for tracking energy consumption over time.

Another big functionality of this system is the real-time visualization. The Python script uses Matplotlib to plots the sensor data in a XY-grid, the GUI updates it very second, reflecting the latest received data values.

In addition to real-time updates, the system includes functionality for manual data export. A separate thread is constantly active for keyboard input. When the user types ”save” into the terminal, the system captures the current state of the heatmap and saves it as a PNG image. At the same time, all sensor readings are being logged to a timestamped CSV file in a directory. On exit (keyboard interrupt), the system performs a final export to ensure all data is preserved.

5.2.3. Real-time data visualization

By connecting to the server running on the host PC, the transmitted data from ESP332-B can be received and processed in real time. Once the TCP server is fully established, the host continuously listens the incoming data packets from the ESP32-B client. These packets contain temperature, humidity, and air quality. They are parsed and immediately fed into a visualization interface using Python.

By using Python code in section A.5, separate heatmap-style plots are generated for each environmental parameter. These plots update every one seconds to reflect the most recent data collected from the ESP32-B module. The X and Y axes correspond to the robot’s spatial coordinates as it moves, allowing the system to map environmental measurements across different physical locations in real time.

Each grid cell in the plots shows a numerical value at a given (X,Y) location, with the cell color intensity indicating the magnitude of the measurement:

- The **temperature map** uses a yellow-red gradient colormap, where darker red represents higher temperatures.
- The **humidity map** uses a blue gradient colormap, with darker blue represents higher humidity levels.
- The **Air-quality map** uses a also yellow-red gradient colormap, with darker red represents higher VOC-index values.

This form of visualization provides an intuitive way to observe spatial patterns in the environment that the robot is monitoring. However, because the fully integration was not completed as expected, the corresponding X and Y coordinates could not be properly pushed to the code. As a result, an example of these graphs is shown in Figure 5.7, where each new data point is generated as the robot’s X and Y coordinates increment by one unit in each iteration.

Additionally, the system offers user interaction:

- The client can manually save the plots and the corresponding CSV file, the user can type the command ”save” in the Python terminal.
- To exit and save automatically, the user simply closes the plot window. This action triggers the program to store both the current plots and all accumulated data in CSV format for future analysis.

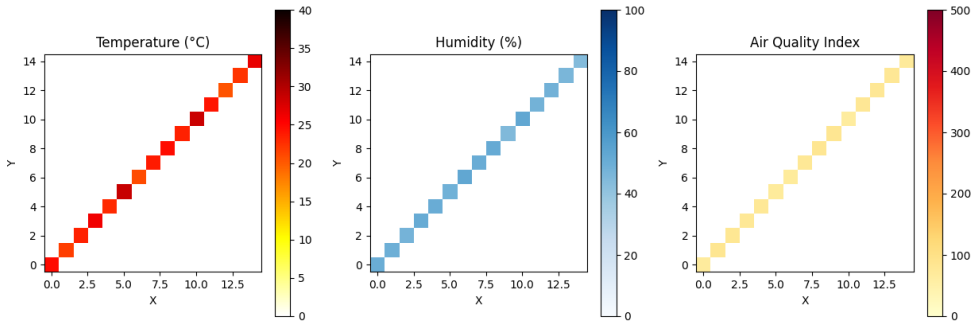


Figure 5.7: temperature, humidity and air quality heatmaps

5.3. Sensing Module Implementation

5.3.1. Finite State Machine

The robot's obstacle avoidance controller is organized as a finite-state five-state machine in Figure 4.5, whose 'hub' is the stop state: depending on the three ultrasonic sensors (left, right, front), the robot either drives straight (forward when all sensors are clear), turns in place (Turn Left if only the right sensor is clear; Turn Right if only the left sensor is clear). Each of these manoeuvres is transient—once the pivot or linear burst is completed, the logic forces the robot back into Stop and holds it there for a fixed five-second dwell. This pause is intentional: it lets the motor's PI speed-control loop settle, eliminates overshoot, and gives the sensors a stable baseline before the next decision cycle. In other words, every left or right turn, reverse move, or forward dash ultimately funnels back through a mandatory five-second stop to ensure the PI controller and the robot's heading are steady before new sensor data can trigger the next action. Due to limited time, the team have not successfully completed the final integration. The FSM performance could not be measured yet.

5.3.2. Temperature & Humidity Sensor Testing and Validation

To evaluate environmental sensors, SHT40 (temperature and humidity) and SGP40 (air quality) are connected to the available I²C pins on the two Sapphires. Data was then logged to a CSV file using wireless transmission to a TCP server. Time stamps were recorded in milliseconds, and both sensors sampled data at roughly 0.5 Hz (one reading per two seconds).

The first part of the test was conducted on the SHT40 sensor, the test environment was indoor inside the room, the sensor was placed on the table where a distance of 20 cm further away a digital thermometer is placed. The sensor is set to measure the data every 2 seconds to make the measurement more readable on Wi-Fi output. both the temperature and relative humidity are then measured. The result is shown in Figure 5.8.

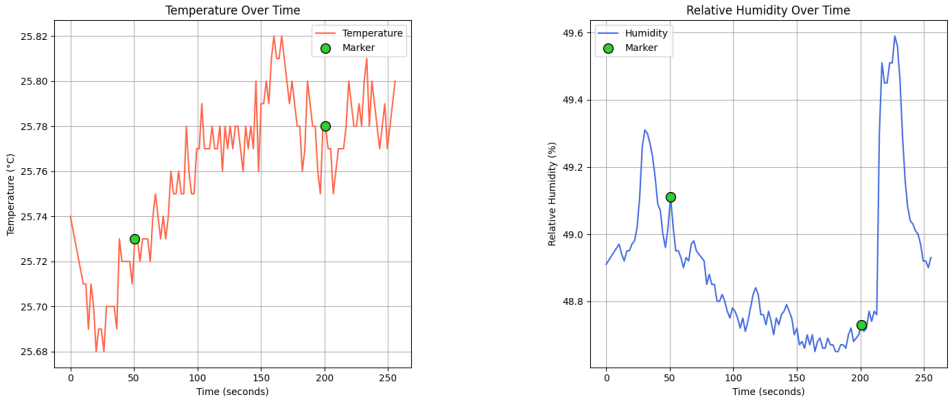


Figure 5.8: Temperature and relative humidity measurements

Looking first at the left graph, the temperature is mainly within the range of 25.68 °C to 25.82 °C. in Figure 5.9, a wired digital thermometer measured the temperature at 25.5 °C, which results in a difference within the 0.3°C range.

Figure 5.9: Fluke T3000 FC Wired Digital Thermometer

The second graph on the right shows the RH(relative humidity). While our group did not have the equipment to directly measure this value, an estimation can be set using the measured temperature and the following formula:

$$RH = \frac{e}{e_s(T)} * 100\% \tag{5.1}$$

From this formula, the e is the actual vapor pressure, for simplicity assume it is constant and has $15.825\ hPa$ as value, and $e_s(T)$ is the saturation vapor pressure at temperature T, shown in Equation 5.2. This formula shows that relative humidity is inversely related to temperature.

$$e_s(T) = 6.112 * exp(\frac{17.67 * T}{T + 243.5}) \tag{5.2}$$

In Figure 5.8, as the temperature drops, the RH increases correspondingly at those spots. Two green markers have been added to both graphs at t = 50 seconds and t = 200 seconds to illustrate how the temperature can effect the RH.

At t = 50 s, the measured temperature is 25.73 °C, using the Equation 5.2.

$$e_s(25.73) \approx 31.7\ hPa \tag{5.3}$$

Assuming using the constant e:

$$RH = \frac{15.825}{31.7} * 100\% \approx 49.9\% \tag{5.4}$$

The measured RH is equal to 49.1 %, with is pretty close to the calculated one. Another point is at t = 200 s, the measured temperature is 25.78 °C. By following the same calculation as before, the RH is calculated as 49.8 %. From the plot, RH is read as 48.7 %, which is also very close to the calculated one.

5.3.3. Air Quality Sensor Testing and Validation

As part of our study, our team also conducted a test using the SGP40 sensor to evaluate indoor air quality. The sensor was placed in the same location as the previous sensor to ensure consistent environmental conditions. In this experiment, data was sampled every 2 seconds, like the previous one. This test has a duration of approximately 8 minutes.

The SGP40 outputs a VOC index, using the Sensiron's onboard VOC algorithm[2]. As shown in Figure 5.10, the VOC index gradually increases during the first 270 seconds of the measurement. This initial rise is expected, the sensor and its internal algorithm require a warm-up phase to adapt to the surrounding air and to set a baseline. After this period, the readings begin to stabilize, indicating that the sensor has completed its initialization and is now ready to read reliable air quality data.

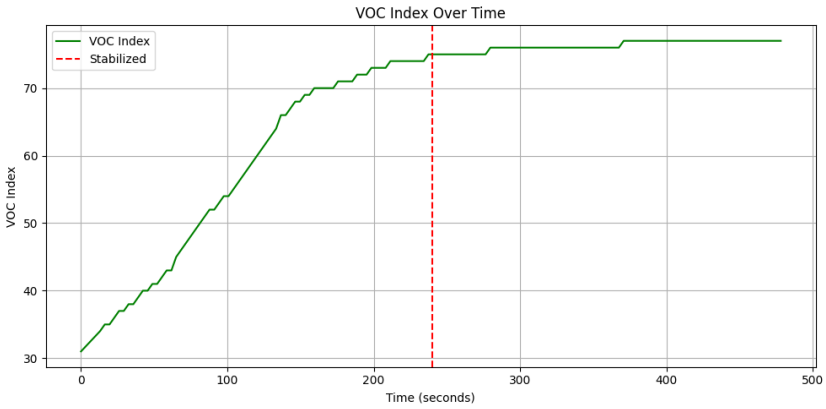


Figure 5.10: VOC-index measurement over a period of 8 minutes

The VOC index value provided by the SGP40 sensor is normalized indicator of indoor air quality. In Figure 5.11, according to the VOC index scale from the SGP40 datasheet [4], a value below 100 is considered to represent clean or healthy indoor air. In our experiment, our result observed a VOC index of approximately 77, which suggests that the air quality in the testing environment was good and free from significant VOC pollution. The scale is designed to be intuitive: value between 0-100 indicate very low VOC presence, 100-200 signal moderate levels, and values above 200 reflect increasingly poor air quality.

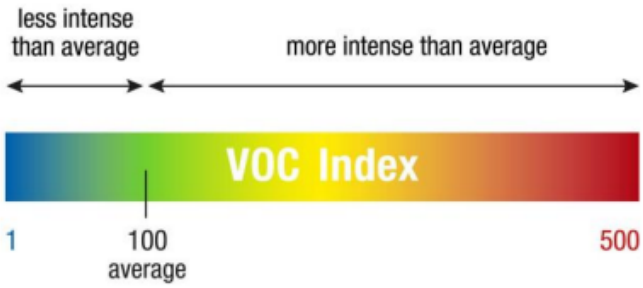


Figure 5.11: Explanation of the VOC index scale[4]

5.4. Hardware Implementation

In this subsection, we are going to go through the hardware implementation related to the Wireless subgroup. The CAN connection, I²C connection and connection via reserved pins will be discussed.

5.4.1. Ultrasonic Distance Sensor on Diamond

The forward-facing ultrasonic sensor is hosted on the Diamond PCB rather than on Ruby or Sapphire for a purely practical reason: Both of the latter boards have exhausted their free GPIOs after allocating lines to the left and right-hand sensors, microphones, I²C bus, status LEDs, and debugging interfaces. Diamond still exposes two spare ADC-capable pins, A1 and A2, as shown in Figure 5.12, assigned by a red rectangle. Shown below, on its reserved header, are ideally suited for the sensor's digital interface.

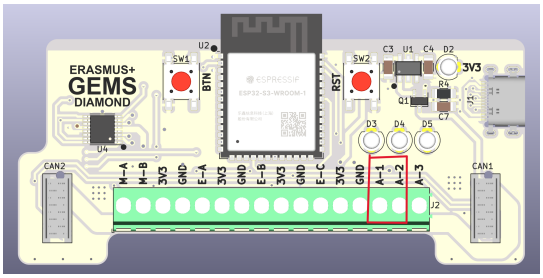


Figure 5.12: Front of Diamond (Motor Drive)

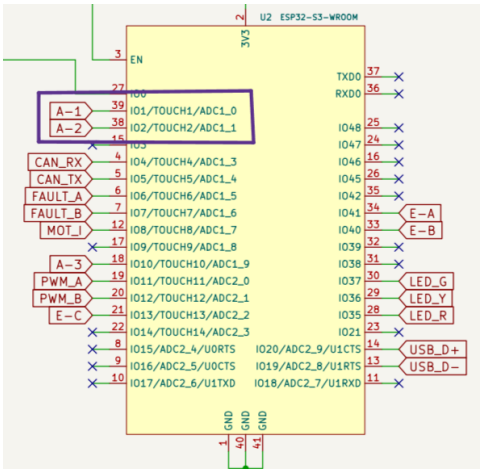


Figure 5.13: Schematic of Diamond (Motor Drive)

At runtime, the ESP32-C3 on Diamond generates a 10 µs logic-high on TRIG, waits for the reflected ECHO pulse, then measures its width with the RMT (Remote Control) peripheral at 40 MHz resolution. The distance d is computed locally using the function shown below:

$$d = \frac{V_{sound}}{2} * t \tag{5.5}$$

where the speed of sound is 343 m/s at room temperature and t is the measured pulse width.

The result, quantized to steps of 1 cm, is packed into a two-byte payload and broadcasts on the CAN bus under frame ID 0x300 (left side) or 0x505 (right side). Sapphire subscribes to that ID, combines the forward distance with the lateral readings it already receives from Ruby, and feeds the distances into its local finite-state machine. If the freshly updated front distance violates the 30 cm safety threshold, Sapphire forces an immediate transition to the Stop (or Turn) state and, after the prescribed five-second dwell, re-evaluates the path.

Putting the front ultrasonic sensor on the Diamond board fixes our lack-of-pins problem without any messy rewiring. We just use two spare pins already present in Diamond's connector, and the distance reading drops straight into the same CAN message list we're using for everything else.

5.4.2. Alternative CAN Connection

The system was designed to link the two CAN buses with a pre-fabricated cable plugged into the dedicated CAN connector. When that cable did not arrive due to a logistics delay, we replaced it with a 'hardwired' direct point-to-point link.

The schematic in Figure 5.14 shows a dedicated 4-pin "CAN Connector" intended to carry the complete bus harness: CAN +, CAN-, + BATT (raw battery rail) and GND, via a pre-fabricated shielded cable.

To replace the missing prefabricated CAN cable, we simply hard-wired Figure 5.15 the two Sapphire boards together. A red jumper carries the battery line (+BATT) from the left board to the right one, and

a black jumper does the same for ground. A twisted blue-and-white pair connects CAN+ (CAN-H) to CAN– (CAN-L), leaving the on-board termination resistors to do their job. Any other pins in the header are left untouched. All four jumpers are soldered in place and protected with heat shrink tubing, giving us the same electrical connection that the cable would have provided, just without the connector.

Figure 5.14: CAN Cable Example

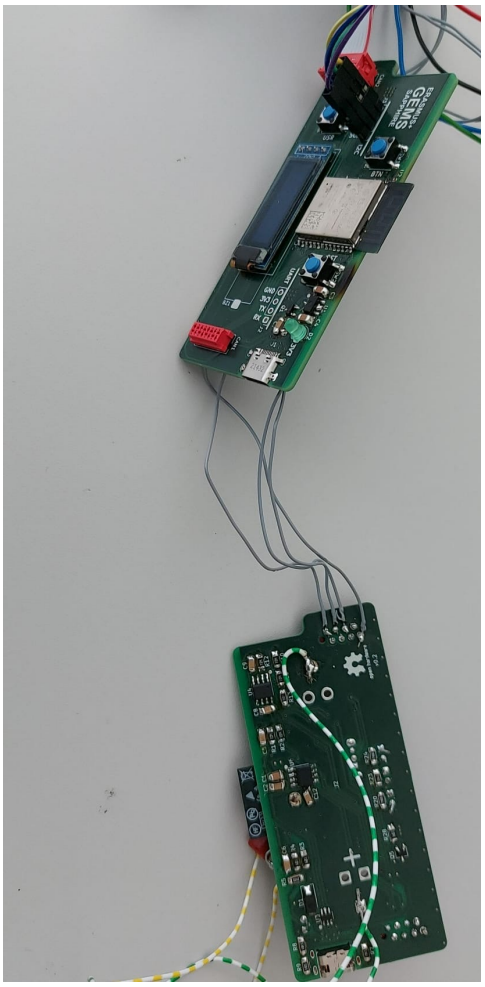


Figure 5.15: Fly-Wiring for CAN replacement

Right now the CAN bus is held together with a tangle of temporary jumpers, but that's only a quick fix. Once the proper cable shows up, we'll pull those makeshift wires, plug in the ready-made connector, and the setup will look tidy and professional again.

5.4.3. I2C

The two-wire bus is brought out on a labelled 4-pin I²C header in the centre of every Sapphire board—pin order (from left to right) is SDA, SCL, +3V3 and GND on a 2.54 mm pitch, as shown in Figure 5.16 assigned by red rectangle. Each header already carries 4.7 kΩ pull-ups (R20, R21) to the 3 V3 rail, so no extra resistors are needed when a sensor is attached. For easy prototyping, we tinned the pads, dropped in short pin headers, and soldered them flush to the PCB; the sensor break-out boards then slide straight onto these pins like miniature "shields".

Two boards share the same I²C bus—one carries the air-quality sensor, the other a temperature-and-humidity sensor. Because each device has its own address, they work side-by-side with no extra wiring or code tweaks. We run the bus at 400 kHz, so even a full data packet is finished in well under a millisecond and never interferes with the CAN traffic. The signals stay clean with jumper leads up to about 20 cm; for anything longer we just twist the two wires together and keep them clear of the motor leads to avoid electrical noise.

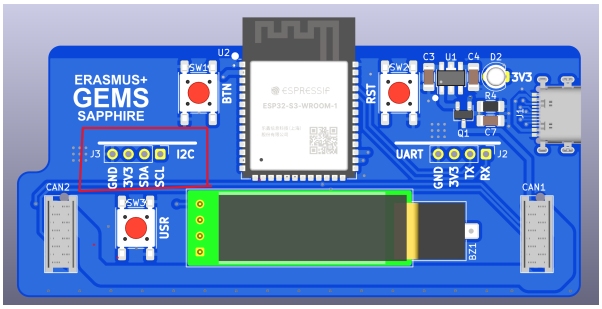


Figure 5.16: Front View of Sapphire

Swapping a sensor is genuinely plug-and-play Figure 5.16: power the board off, lift the little daughter-board off its four-pin header, press the new one on, power back up, and—if the newcomer uses a different I²C address—change a single line in the firmware. No tracks are cut, no other modules are disturbed, and the system is instantly ready for newer or more accurate environmental sensors.

6

Discussion

This project included different elements working together. Even though all of us worked well individually, bringing components together into one fully integrated system did not go as planned. The reason for this was the time constraints we faced, along with delays in receiving some essential components.

On the other side, most of the hardware and software components were designed and tested successfully. The CAN communication system was reliable and helped everything organized and simple. Each message had its own ID and we could prioritize certain types of data when needed. We also left some unused message slots on purpose, in this way it will be much easier to add more sensors or features later on.

The two environmental sensors, especially the SHT40 temperature and humidity sensor, produced accurate reading during testing. We compared the temperature values to a digital thermometer and they were placed very close. For humidity, we didn't have direct comparison device, but used theoretical calculations to estimate expected values, which aligned well with the sensor readings.

The air quality sensor (SGP40) also performed as expected. It needed a few minutes to warm up, which we knew from the documentation. After that, stable readings are provided. The final measured VOC index was in the ”clean air” range, which made sense given the indoor environment.

The software side was one of the important parts of the project. The web server running on the ESP32 made it easy to check live values and control the robot wirelessly. The python script handling Wi-Fi communication and updated the live plots. We were also able to log data, save graphs, and even take snapshots using user commands.

Unfortunately, we did not succeed in combining all the systems to create a fully operational robot with other groups. A big part of this came down to the time, each group spent too long perfecting individual components and did leave enough time to handle fully integration and testing as a whole system. On top of that, some key parts (like the CAN cable) didn't arrive on time, which forced us to make some extra works. These temporary fixes worked to some extent, but they slowed down the process and made it more difficult to debug the final system.

Overall, the results achieved so far have been satisfactory, a solid foundation was established. However, we also gained an important lesson in the value of planning ahead, allocating more time for system integration, considering worst case scenarios, and developing plans for unforeseen issues.

7

Conclusion

By the end of our bachelor's project, the wireless system does everything it should. The project met its primary objective: demonstrating a reliable, low-cost communication layer for a modular robot. A fully functional CAN backbone now links the power, sensing and drive modules; an I²C sub-bus supports plug-and-play environmental sensors; a five-state finite-state machine steers the robot safely around obstacles; and a Wi-Fi web page provides live telemetry and manual control. These elements collectively prove that an ESP32-based architecture can satisfy real-time requirements while remaining affordable and easy to reproduce in an educational setting.

The first point, our goal is not clear. At the start, we never settled down on exactly what "done" looked like, so we kept changing utilities and redoing work. We have changed the communication protocol from Bluetooth to Wi-Fi and wireless to cabled CAN communication. For the next time, we should draft a single-page Definition of Done that lists every final deliverable as concise, measurable bullets. Pair each bullet with its verification method, specify immutable interface details, set deadlines, and record excluded features. Review mid-project, freeze revisions, and use the sheet as the only benchmark for progress.

The second point is the communication within the group. This caused a lot of delay in the progress. For instance, the motor, power and wireless groups each did their own thing for weeks, and we only tried to bolt everything together near the deadline, leaving almost no time to fix bugs and continue with Should-Have. On the other hand, logistics problems are also caused by miscommunication. We had to replace the missing prefab cable with loose jumpers; they work, but they look rough and could short out. For the next time, to avoid encountering organisational and wiring problems, two measures should be taken from the outset. First, use shared digital task boards (Google To Do List) and schedule mandatory bi-weekly full-system integration meetings; these practices allow for early detection of cross-discipline interface failures, assign clear ownership, and ensure completion in the next sprint. Second, prefabricated CAN harnesses should be ordered during the initial procurement phase to ensure hardware reliability. If temporary jumpers are required, they must be heat-shrink sleeved, fitted with strain relief anchors, and documented with colour-coded pinout diagrams. Together, these measures will simplify the integration process, reduce potential failures, and produce a safer, more professional wiring layout.

At the end of this thesis, we will conclude everything we did. We have built a reliable and trustworthy communication system. Even though they are not perfect, they satisfied our PoR.

References

[1] David Abreu et al. “Low-Cost Ultrasonic Range Improvements for an Assistive Device”. In: *Sensors* 21.12 (2021), p. 4250. DOI: 10.3390/s21124250. URL: https://www.mdpi.com/1424-8220/21/12/4250.

[2] Adafruit. *Adafruit SGP40 Arduino Library*. https://github.com/adafruit/Adafruit_SGP40. Accessed: 2025-06-14. 2021.

[3] AfterAcademy. “What is a TCP 3-way handshake process?” In: *AfterAcademy* (2020). Accessed: 2025-06-06. URL: https://afteracademy.com/blog/what-is-a-tcp-3-way-handshake-process/.

[4] Sensirion AG. *SGP40 Datasheet – Indoor Air Quality Sensor for VOC Measurements*. https://www.farnell.com/datasheets/4020671.pdf. Accessed: 2025-06-06. 2021.

[5] Sensirion AG. *SHT4x Datasheet – Digital Humidity Sensor*. https://www.farnell.com/datasheets/3512208.pdf. Accessed: 2025-06-06. 2022.

[6] Geoffrey Brown. “Discovering the STM32 microcontroller”. In: *Cortex* 3.34 (2012), p. 64.

[7] Espressif Systems. *ESP-NOW API Reference - ESP32 - ESP-IDF Programming Guide*. https://docs.espressif.com/projects/esp-idf/en/latest/esp32/api-reference/network/esp_now.html. Accessed: 2025-06-13. 2025.

[8] Espressif Systems. *ESP32–C3–WROOM–02 & WROOM–02U Datasheet*. Datasheet. Version v1.5. Shanghai, China: Espressif Systems, Apr. 2025. URL: https://www.espressif.com/sites/default/files/documentation/esp32-c3-wroom-02_datasheet_en.pdf.

[9] GeeksforGeeks. *TCP 3-Way Handshake Process*. Accessed: 2025-06-06. 2024. URL: https://www.geeksforgeeks.org/tcp-3-way-handshake-process/.

[10] *GEMS Erasmus+ — gems-erasmus.eu*. https://gems-erasmus.eu/. [Accessed 15-06-2025].

[11] C L Glennie et al. “Geodetic imaging with airborne LiDAR: the Earth’s surface revealed”. In: *Reports on Progress in Physics* 76.8 (July 2013), p. 086801. DOI: 10.1088/0034-4885/76/8/086801. URL: https://dx.doi.org/10.1088/0034-4885/76/8/086801.

[12] handmade0octopus. *ESP32 TWAI CAN*. https://github.com/handmade0octopus/ESP32-TWAI-CAN. 2015.

[13] Jolle W Jolles. “Broad-scale applications of the Raspberry Pi: A review and guide for biologists”. In: *Methods in Ecology and Evolution* 12.9 (2021), pp. 1562–1579.

[14] Yutaka Kaneda and Juro Ohga. “Adaptive microphone-array system for noise reduction”. In: *IEEE Transactions on Acoustics, Speech, and Signal Processing* 34.6 (1986), pp. 1391–1400.

[15] Hermann Kopetz, Wilfried Elmenreich, and Christoph Mack. “A comparison of lin and ttp/a”. In: *2000 IEEE International Workshop on Factory Communication Systems. Proceedings (Cat. No. 00TH8531)*. IEEE. 2000, pp. 99–107.

[16] Neha R Laddha and AP Thakare. “A review on serial communication by UART”. In: *International journal of advanced research in computer science and software engineering* 3.1 (2013).

[17] Hiram RayoTorres-Rodríguez. “Design and development of a driver based on J2716 standard for data transmission on high electrical noise environments”. In: (2016).

[18] Emily J Smith et al. “Enhancing your everyday sight: An ultrasonic visual aid”. In: *Frontiers in Biomedical Devices*. Vol. 84815. American Society of Mechanical Engineers. 2022, V001T04A001.

[19] Jonathan Valdez and Jared Becker. “Understanding the I2C bus”. In: *Texas instruments* (2015), p. 8.

[20] Donald R Wehner. “High resolution radar”. In: *Norwood* (1987).

[21] Yi-Hung Wei et al. “RT-WiFi: Real-Time High-Speed Communication Protocol for Wireless Cyber-Physical Control Applications”. In: *2013 IEEE 34th Real-Time Systems Symposium*. 2013, pp. 140–149. DOI: 10.1109/RTSS.2013.22.

A

Source Code

A.1. Right Sapphire

```
#include <ESP32-TWAI-CAN.hpp>
#include <Wire.h>
#include <Adafruit_SSD1306.h>
#include <Adafruit_GFX.h>
#include <Adafruit_SGP40.h>

// === CAN Pins ===
#define CAN_RX 4
#define CAN_TX 5

// === Display Settings ===
#define WIDTH 128
#define HEIGHT 32
#define SDA 11
#define SCL 12


Adafruit_SSD1306 display(WIDTH, HEIGHT, &Wire, -1);
Adafruit_SGP40 sgp;  // SGP40 object

// === Data from CAN ===
float voltageRight = 0.0;
float currentRight = 0.0;
uint8_t ledIndexRight = 0;
float socRight = 0.0;
float distanceRight = 0.0;
float speedRight = 0.0; //CAN
float dutyCycleRight = 0.0;
float dutyCycleLeft = 0.0;
float tempEmeraldRight = 0.0; // CAN
uint8_t checkState = 0;
// === Threshold ===
int threshold = 30;
unsigned long lastFSMUpdate = 0;

// === VOC and TEMP and HUMI ===
bool key = false;
int N = 0;
unsigned long lastUpdate = 0;

// Declare global variables for temp and humi
float Temperature = 0.0;
float Humidity = 0.0;

uint16_t VOC = 0;


```

```
48 void updateOLED() {
49   display.clearDisplay();
50   display.setTextSize(1);
51   display.setTextColor(SSD1306_WHITE);
52   display.setCursor(0, 0);
53   display.print("V: "); display.print(voltageRight, 2); display.print("V ");
54   display.print("I: "); display.print(currentRight, 2); display.println("A");
55   // display.print("Dist: "); display.print(distance); display.print("cm ");
56   display.print("LED: "); display.println(ledIndexRight);
57   display.print("SOC:"); display.println(socRight);
58   display.print("VOC:"); display.println(VOC);
59   display.display();
60 }
61
62 void setup() {
63   Serial.begin(115200);
64   Wire.begin(SDA, SCL);
65
66   // Search for existing I2c Address
67   Serial.println("I2C Scanner");
68   for (uint8_t addr = 1; addr < 127; addr++) {
69     Wire.beginTransmission(addr);
70     if (Wire.endTransmission() == 0) {
71       Serial.print("I2C device found at 0x");
72       Serial.println(addr, HEX);
73       delay(10);
74     }
75   }
76   Serial.println("Scan done.");
77   if (!display.begin(SSD1306_SWITCHCAPVCC, 0x3C)) {
78     Serial.println(F("OLED init failed"));
79     while (1);
80   }
81
82   display.clearDisplay();
83   display.setTextSize(1);
84   display.setCursor(0, 0);
85   display.println("Starting...");
86   display.display();
87
88   // === CAN Init ===
89   ESP32Can.setPins(CAN_TX, CAN_RX);
90   ESP32Can.setSpeed(ESP32Can.convertSpeed(500));
91   ESP32Can.setRxQueueSize(10);
92   ESP32Can.setTxQueueSize(10);
93
94   if (!ESP32Can.begin()) {
95     Serial.println("CAN init failed");
96     while (1);
97   }
98   Serial.println("CAN Ready");
99
100  // SGP40 init
101  if (!sgp.begin()) {
102    Serial.println("SGP40 not found :(");
103    while (true);  // Freeze if sensor not found
104  }
105  Serial.println("SGP40 ready!");
106
107  Serial.println("Ready to send command to right Sapphire");
108 }
109
110 //not used, do not want to change structure
111 void sendCANCommand(uint8_t state) {
112   CanFrame frame = { 0 };
113   frame.identifier = 0x000;
114   frame.extd = 0;
115   frame.data_length_code = 1;
116   frame.data[0] = state;
117
118   ESP32Can.writeFrame(frame);
```

```
}

void sendLEDCANCommand(uint8_t ledIndex) {
  CanFrame frame = { 0 };
  frame.identifier = 0x500;
  frame.extd = 0;
  frame.data_length_code = 2;
  frame.data[0] = 0x01;
  frame.data[1] = ledIndex;

  ESP32Can.writeFrame(frame);
}

void sendcheckCommand(uint8_t checkState) {
  CanFrame frame = { 0 };
  frame.identifier = 0x720;
  frame.extd = 0;
  frame.data_length_code = 2;
  frame.data[0] = 0x01;
  frame.data[1] = checkState;

  ESP32Can.writeFrame(frame);
}


// uint8_t state(float distanceLeft, float distanceRight, float distanceFront){
//   // === FSM based on distance sensors every 1s ===
//   unsigned long now = millis();
//   if (now - lastFSMUpdate >= 1000) {
//     lastFSMUpdate = now;
//     bool L = distanceLeft < threshold;
//     bool R = distanceRight < threshold;
//     bool F = distanceFront < threshold;

//     uint8_t action = 0; // 0 = stop, 1 = forward, 2 = backward, 3 = right, 4 = left, 5 = left/right

//     if (!L && !R && !F) action = 1;
//     else if (!L && !R && F) action = 5;
//     else if (!L && R && F) action = 4;
//     else if (!L && R && !F) action = 4;
//     else if (L && !R && F) action = 3;
//     else if (L && !R && !F) action = 3;
//     else if (L && R && !F) action = 3;
//     else if (L && R && F) action = 5;
//     else action = 0;
//   }
//   return action;
// }
void sendDriveCommand(uint8_t command) {
  // command: 0 = stop, 1 = forward, 2 = backward, 3 for right, 4 for left, 5 for left/right
  CanFrame frame = { 0 };
  frame.identifier = 0x710;
  frame.extd = 0;
  frame.data_length_code = 6;
  frame.data[0] = command;
  ESP32Can.writeFrame(frame);

  if (ESP32Can.writeFrame(frame)) {
    Serial.print("Sent Drive Command to Diamond: ");
    Serial.println(command);
  } else {
    Serial.println("Failed to send drive command");
  }
}

void sendDriveduty(float command) {
  CanFrame frame = { 0 };
  frame.identifier = 0x730;
  frame.extd = 0;
  frame.data_length_code = 6;
```

```
  frame.data[0] = command;
  ESP32Can.writeFrame(frame);

  if (ESP32Can.writeFrame(frame)) {
    Serial.print("Sent Drive Command to Diamond: ");
    Serial.println(command);
  } else {
    Serial.println("Failed to send drive command");
  }
}

void loop() {
  VOC = sgp.measureVocIndex();
  Serial.print("VOC: ");
  Serial.println(VOC);
  unsigned long now = millis();

  // === CAN Receive ===
  CanFrame frame;
  while (ESP32Can.readFrame(frame, 10)) {
    switch (frame.identifier) {
      case 0x405:  // voltageLeft, currentRight, socLeft
        if (frame.data_length_code >= 6) {
          uint16_t current_mA = (frame.data[3] << 8) | frame.data[2];
          uint16_t voltage_mV = (frame.data[5] << 8) | frame.data[4];
          currentRight = current_mA / 1000.0;
          voltageRight = voltage_mV / 1000.0;
          socRight = frame.data[1];
        }
        break;

      case 0x500:  // ledIndexLeft (receive visual feedback)
        if (frame.data_length_code == 2) {
          ledIndexRight = frame.data[1];
        }
        break;

      case 0x705:  // Speed
        if (frame.data_length_code == 2) {
          speedRight = frame.data[1];
        }
        break;

      // case 0x700:  // Duty cycle check
      //   if (frame.data_length_code == 2) {
      //     ledIndexRight = frame.data[1];
      //   }
      //   break;
      case 0x315:  // checkstate
        if (frame.data_length_code == 2) {
          checkState = frame.data[1];
        }
        sendcheckCommand(checkState);
        break;
      // case 0x300:  // distanceLeft
      //   if (frame.data_length_code == 4) {
      //     memcpy(&distanceLeft, &frame.data[0], sizeof(float));
      //     Serial.print("distance Left: ");
      //     Serial.print(distanceLeft);
      //     Serial.println("cm");
      //   }
      //   break;

      case 0x505:  // distanceRight
        if (frame.data_length_code == 4) {
          memcpy(&distanceRight, &frame.data[0], sizeof(float));
          Serial.print("distance right: ");
          Serial.print(distanceRight);
          Serial.println("cm");
        }
        break;
```

```
260
261       case 0x700:  // duty_cycle right
262         if (frame.data_length_code == 4) {
263           memcpy(&dutyCycleRight, &frame.data[0], sizeof(float));
264           Serial.print("Duty Cycle: ");
265           Serial.println(dutyCycleRight);
266           sendDriveduty(dutyCycleRight);
267         }
268         break;
269
270       case 0x220:  // duty left
271         if (frame.data_length_code == 4) {
272           memcpy(&dutyCycleLeft, &frame.data[0], sizeof(float));
273           Serial.print("distance front: ");
274           Serial.print(dutyCycleLeft);
275           Serial.println("cm");
276         }
277         break;
278       default:
279         Serial.print("Unknown CAN ID: 0x");
280         Serial.println(frame.identifier, HEX);
281         break;
282     }
283   }
284 }
```

A.2. Left Sapphire

```
1     WiFiClient client = server.available();
2 if (client) {
3   Serial.println("Client connected");
4   while (!client.available()) delay(1);
5   String request = client.readStringUntil('\r');
6   client.flush();
7
8   // === Command Handling ===
9   if (request.indexOf("/on") != -1) {
10    sendCANCommand(1);
11  } else if (request.indexOf("/off") != -1) {
12    sendCANCommand(0);
13  } else if (request.indexOf("/led?val=") != -1) {
14    int valIndex = request.indexOf("/led?val=") + 9;
15    uint8_t val = request.substring(valIndex, request.indexOf(' ', valIndex)).toInt();
16    if (val <= 6) {
17      sendLEDCANCommand(val);
18    }
19  } else if (request.indexOf("/forward") != -1) {
20    action = 1;
21    sendDriveCommand(action);
22  } else if (request.indexOf("/backward") != -1) {
23    action = 2;
24    sendDriveCommand(action);
25  } else if (request.indexOf("/stop") != -1) {
26    action = 0;
27    sendDriveCommand(action);
28  }
29
30  // === Web Page ===
31  client.println("HTTP/1.1 200 OK");
32  client.println("Content-Type: text/html");
33  client.println("Connection: close");
34  client.println();
35  client.println("<!DOCTYPE html><html><head><meta http-equiv='refresh' content='2'>");
36  client.println("<title>GEMS Sapphire</title></head><body style='text-align:center;'>");
37  client.println("<h1>GEMS Sapphire Status</h1>");
38  client.print("<p><b>Voltage:</b> "); client.print(voltageLeft); client.println(" V</p>");
39  client.print("<p><b>Current:</b> "); client.print(currentLeft); client.println(" A</p>");
40  client.print("<p><b>LED Index:</b> "); client.print(ledIndexLeft); client.println("</p>");
41  client.print("<p><b>SoC:</b> "); client.print(socLeft); client.println("</p>");
42  client.print("<p><b>Temperature:</b> "); client.print(Temperature); client.println("</p>");
```

```
  client.print("<p><b>Humidity:</b> "); client.print(Humidity); client.println(" </p>");
  // client.print("<p><b>VOC:</b> "); client.print(VOC); client.println("</p>");

  client.println("<br><a href='/on'><button style='padding:10px;'>Turn ON Emerald</button></a>");
  client.println("<a href='/off'><button style='padding:10px;'>Turn OFF Emerald</button></a>");

  client.println("<hr><form action='/led'>");
  client.println("LED Index (0-6): <input name='val' type='number' min='0' max='6'>");
  client.println("<input type='submit' value='Set LED'>");
  client.println("</form>");

  client.println("<hr><h2>Drive Control</h2>");
  client.println("<a href='/forward'><button style='padding:10px;'>Forward</button></a>");
  client.println("<a href='/backward'><button style='padding:10px;'>Backward</button></a>");
  client.println("<a href='/stop'><button style='padding:10px;'>Stop</button></a>");
  client.println("</body></html>");
  client.stop();
}
}
```

A.3. Left Ruby

```
#include <WiFi.h>
#include <ESP32-TWAI-CAN.hpp>
#include "Adafruit_SHT4x.h"

// === Can Pins ===
#define CAN_RX 4
#define CAN_TX 5

// === Other pins ===
#define MIC_L 0
#define MIC_R 1
#define LED_A 3
#define LED_B 7
#define LED_C 10
#define BTN 9
#define US_TRIG 20
#define US_ECHO 21

#define MIC_GAIN 123

#define H 1
#define L 0
#define Z -1


// === WiFi Settings ===
const char* ssid = "GEMS_Sapphire";
const char* password = "equalequal";
const char* host = "192.168.3.4";  // Replace with your PC's IP address

const uint16_t port = 12345;

// === distance variables ===
long duration;
float distanceLeft;

// === Data from CAN ===
float temperature = 0.0;
float temp = 0.0;    // temp from the Emerald
float humidity = 0.0;
uint8_t VOC = 0;
float voltage = 0;
float current = 0;
float SOC = 0;
unsigned long ts = 0;

```

```
// === Robot Position ===
int x = 0;
int y = 0;
float velocity = 0.1; // meters per second (example)
int direction = 1;    // 0=static, 1=forward, 2=backward, 3=right, 4=left

WiFiClient client;
unsigned long lastSendTime = 0;
const unsigned long sendInterval = 5000; // 5 seconds

void setup() {
  Serial.begin(115200);

  // === Ruby general pins ===
  pinMode(LED_A, OUTPUT);
  pinMode(LED_B, OUTPUT);
  pinMode(LED_C, OUTPUT);
  pinMode(US_TRIG, OUTPUT);
  pinMode(US_ECHO, INPUT);

  digitalWrite(LED_A, LOW);
  digitalWrite(LED_B, LOW);
  digitalWrite(LED_C, LOW);

  delay(300);
  Serial.println("Ultrasonic LED distance display");

  // === Using this code to check available services ===
  Serial.println("Scanning for networks...");

  int n = WiFi.scanNetworks();
  if (n == 0) {
    Serial.println("No networks found.");
  } else {
    Serial.println("Networks found:");
    for (int i = 0; i < n; ++i) {
      Serial.printf("%d: %s (%d dBm)\n", i + 1, WiFi.SSID(i).c_str(), WiFi.RSSI(i));
      delay(10);
    }
  }

    // === Connect to WiFi ===
  WiFi.begin(ssid, password);
  Serial.print("Connecting to WiFi");
  while (WiFi.status() != WL_CONNECTED) {
    delay(500);
    Serial.print(".");
  }
  Serial.println("\nWiFi connected!");
  Serial.println(WiFi.localIP());

  Serial.print("Pinging host: ");
  Serial.println(host);
  if (WiFi.status() == WL_CONNECTED) {
  Serial.println("WiFi is connected.");
  } else {
  Serial.println("WiFi not connected.");
  }

  // === Can init ===
  ESP32Can.setPins(CAN_TX, CAN_RX);
  ESP32Can.setTxQueueSize(10);
  ESP32Can.setRxQueueSize(10);
  ESP32Can.setSpeed(ESP32Can.convertSpeed(500));

  if (!ESP32Can.begin()) {
    Serial.println("CAN init failed");
    while (1);
  }
    Serial.println("CAN Ready");
}
```

```
118
119 void loop() {
120   // === Check the connection ===
121   if (!client.connected()) {
122     Serial.println("Connecting to server...");
123     if (client.connect(host, port)) {
124       Serial.println("Connected to Python server!");
125     } else {
126       Serial.println("Connection failed.");
127       delay(5000);
128       return;
129     }
130   }
131
132   // === distance determination ===
133   distanceLeft = getDistance();
134   int ledIndex = mapDistanceToLED(distanceLeft);
135   single_LED(ledIndex);
136
137     // === Send CAN ===
138   distanceLeft(distanceLeft);
139   ledIndexLeft(ledIndex);
140
141   // delay(1000);
142    // === CAN receive ===
143   CanFrame frame;
144   while (ESP32Can.readFrame(frame, 10)) {
145       if (frame.identifier == 0x330 && frame.data_length_code == 8) {
146
147         memcpy(&temperature, &frame.data[0], sizeof(float));
148         memcpy(&humidity, &frame.data[4], sizeof(float));
149
150         Serial.print("Temperature: ");
151         Serial.print(temperature);
152         Serial.print(" C, Humidity: ");
153         Serial.print(humidity);
154         Serial.println(" %");
155       }
156     if (frame.identifier == 0x102 && frame.data_length_code >= 4) {
157       memcpy(&ts, &frame.data[0], sizeof(unsigned long));
158
159       Serial.print("Timestamps: ");
160       Serial.print(ts);
161       Serial.println(" ms");
162     }
163
164     if (frame.identifier == 0x110 && frame.data_length_code >= 4) {
165       memcpy(&SOC, &frame.data[0], sizeof(float));
166
167       Serial.print("SOC: ");
168       Serial.print(SOC);
169       Serial.println(" %");
170     }
171
172     if (frame.identifier == 0x100 && frame.data_length_code >= 8) {
173       memcpy(&current, &frame.data[0], sizeof(float));
174       memcpy(&voltage, &frame.data[4], sizeof(float));
175
176       Serial.print("Voltage ");
177       Serial.print(voltage);
178       Serial.print(" mV : Current: ");
179       Serial.print(current);
180       Serial.println(" mA");
181     }
182     if (frame.identifier == 0x106 && frame.data_length_code >= 4) {
183       memcpy(&temp, &frame.data[0], sizeof(float));
184
185       Serial.print("Temp Emerald ");
186       Serial.print(temp);
187       Serial.println(" C");
188     }
```

```
  }


  x = x + 1;
  y = y + 1;

  // char data[64];
  char data_power[64];

  // Data for temperature and humidity
  snprintf(data, sizeof(data), "%d,%d,%.2f,%.2f\n", x, y, temperature, humidity);
  client.print(data);
  Serial.print("Sent: ");
  Serial.println(data);

  // Format voltage, current, and SoC data
  snprintf(data_power, sizeof(data_power), "P, %lu, %.2f, %.2f, %.2f, %.2f\n", ts, SOC,
      voltage, current, temp);
  client.print(data_power);
  Serial.print("Power: ");
  Serial.println(data_power);

delay(3000);

}

// === equation for distance ===
float getDistance() {
  digitalWrite(US_TRIG, LOW);
  delayMicroseconds(2);
  digitalWrite(US_TRIG, HIGH);
  delayMicroseconds(10);
  digitalWrite(US_TRIG, LOW);

  duration = pulseIn(US_ECHO, HIGH, 30000); // timeout at ~5m
  float distance = duration * 0.034 / 2;    // cm
  return distance;
}

// === map distance to LED ===
int mapDistanceToLED(float d) {
  if (d < 10) return 0;
  else if (d < 20) return 1;
  else if (d < 30) return 2;
  else if (d < 40) return 3;
  else if (d < 50) return 4;
  else if (d < 60) return 5;
  else return 6;
}

void single_LED(int n) {
  switch (n) {
    case 0: setCharlieplex(Z, Z, Z); break;
    case 1: setCharlieplex(L, H, Z); break;
    case 2: setCharlieplex(L, Z, H); break;
    case 3: setCharlieplex(Z, L, H); break;
    case 4: setCharlieplex(H, L, Z); break;
    case 5: setCharlieplex(H, Z, L); break;
    case 6: setCharlieplex(Z, H, L); break;
  }
}

// === Set Charlieplex ===
void setCharlieplex(int A, int B, int C) {
  setCharlieplexPin(LED_A, A);
  setCharlieplexPin(LED_B, B);
  setCharlieplexPin(LED_C, C);
}

void setCharlieplexPin(int X, int S) {
  switch (S) {
```

```
    case L: pinMode(X, OUTPUT); digitalWrite(X, LOW); break;
    case H: pinMode(X, OUTPUT); digitalWrite(X, HIGH); break;
    case Z: pinMode(X, INPUT); break;
  }
}

// === send CAN distance ===
void distanceLeft(float distance){
  CanFrame frame = {0};
  frame.identifier = 0x205;
  frame.extd = 0;
  frame.data_length_code = 4;

  // Convert float to 4 bytes
  memcpy(&frame.data[0], &distance, sizeof(float));

  ESP32Can.writeFrame(frame);
}

void ledIndexLeft(uint8_t ledIndex){
  CanFrame frame = {0};
  frame.identifier = 0x200;
  frame.extd = 0;
  frame.data_length_code = 1;

  // Convert float to 4 bytes
  frame.data[0] = ledIndex;

  ESP32Can.writeFrame(frame);
}
```

A.4. Right Ruby

```
#include <ESP32-TWAI-CAN.hpp>

#define LED_A 3
#define LED_B 7
#define LED_C 10
#define CAN_RX 4
#define CAN_TX 5
#define BTN 9
#define US_TRIG 20
#define US_ECHO 21

#define MIC_GAIN 123

#define H 1
#define L 0
#define Z -1

long duration;
float distanceRight;

void setup() {
  Serial.begin(115200);


  pinMode(LED_A, OUTPUT);
  pinMode(LED_B, OUTPUT);
  pinMode(LED_C, OUTPUT);
  pinMode(US_TRIG, OUTPUT);
  pinMode(US_ECHO, INPUT);

  digitalWrite(LED_A, LOW);
  digitalWrite(LED_B, LOW);
  digitalWrite(LED_C, LOW);

  delay(3000);
  Serial.println("Ultrasonic LED distance display");

```

```
38   // === Can init ===
39   ESP32Can.setPins(CAN_TX, CAN_RX);
40   ESP32Can.setTxQueueSize(10);
41   ESP32Can.setRxQueueSize(10);
42   ESP32Can.setSpeed(ESP32Can.convertSpeed(500));
43
44   if (!ESP32Can.begin()) {
45     Serial.println("CAN init failed");
46     while (1);
47   }
48     Serial.println("CAN Ready");
49
50
51 }
52
53
54 void loop() {
55
56   // === distance determination ===
57   distanceRight = getDistance();
58   Serial.print("Distance right: ");
59   Serial.print(distanceRight);
60   Serial.println(" cm");
61
62   uint8_t ledIndex = mapDistanceToLED(distanceRight);
63   single_LED(ledIndex);
64
65   // === Send CAN ===
66   distanceRight(distanceRight);
67   ledIndexRight(ledIndex);
68
69   delay(1000);
70 }
71
72
73 float getDistance() {
74   digitalWrite(US_TRIG, LOW);
75   delayMicroseconds(2);
76   digitalWrite(US_TRIG, HIGH);
77   delayMicroseconds(10);
78   digitalWrite(US_TRIG, LOW);
79
80   duration = pulseIn(US_ECHO, HIGH, 30000); // timeout at ~5m
81   float distance = duration * 0.034 / 2;    // cm
82   return distance;
83 }
84
85 int mapDistanceToLED(float d) {
86   if (d < 10) return 0;
87   else if (d < 20) return 1;
88   else if (d < 30) return 2;
89   else if (d < 40) return 3;
90   else if (d < 50) return 4;
91   else if (d < 60) return 5;
92   else return 6;
93 }
94
95 void single_LED(int n) {
96   switch (n) {
97     case 0: setCharlieplex(Z, Z, Z); break;
98     case 1: setCharlieplex(L, H, Z); break;
99     case 2: setCharlieplex(L, Z, H); break;
100    case 3: setCharlieplex(Z, L, H); break;
101    case 4: setCharlieplex(H, L, Z); break;
102    case 5: setCharlieplex(H, Z, L); break;
103    case 6: setCharlieplex(Z, H, L); break;
104  }
105 }
106
107 void setCharlieplex(int A, int B, int C) {
108   setCharlieplexPin(LED_A, A);
```

```
  setCharlieplexPin(LED_B, B);
  setCharlieplexPin(LED_C, C);
}

void setCharlieplexPin(int X, int S) {
  switch (S) {
    case L: pinMode(X, OUTPUT); digitalWrite(X, LOW); break;
    case H: pinMode(X, OUTPUT); digitalWrite(X, HIGH); break;
    case Z: pinMode(X, INPUT); break;
  }
}

// === send CAN distance ===
void distanceRight(float distance){
  CanFrame frame = {0};
  frame.identifier = 0x505;
  frame.extd = 0;
  frame.data_length_code = 4;

  // Convert float to 4 bytes
  memcpy(&frame.data[0], &distance, sizeof(float));

  ESP32Can.writeFrame(frame);
}

void ledIndexRight(uint8_t ledIndex){
  CanFrame frame = {0};
  frame.identifier = 0x500;
  frame.extd = 0;
  frame.data_length_code = 1;

  // Convert float to 4 bytes
  frame.data[0] = ledIndex;

  ESP32Can.writeFrame(frame);
}
```

A.5. Python script to process sensor data

```
"""
Sensor Network Visualizer & Logger
----------------------------------

This Python script sets up a server application that receives sensor data over a TCP socket,
    processes it in real time, logs it into CSV files, and displays it graphically using
    matplotlib.

Key Features:
- Listens for incoming sensor data on a specified host and port.
- Supports three types of environmental data per grid coordinate:
    - Temperature (°C)
    - Humidity (%)
    - Air Quality Index (AQI)
- Also logs power system telemetry:
    - Timestamp, State of Charge (SoC), Voltage, Current, Internal Temperature
"""


import socket
import matplotlib.pyplot as plt
import numpy as np
import threading
import csv
import os
from datetime import datetime

# === Configuration ===
host = "0.0.0.0"
port = 12345
map_size = 15
```

```

temperature_map = [[None for _ in range(map_size)] for _ in range(map_size)]
humidity_map = [[None for _ in range(map_size)] for _ in range(map_size)]
air_quality_map = [[None for _ in range(map_size)] for _ in range(map_size)]
lock = threading.Lock()
save_requested = threading.Event()

# === General CSV Setup ===
timestamp = datetime.now().strftime("%Y%m%d_%H%M%S")
directory = "data"
os.makedirs(directory, exist_ok=True)

# === Sensing CSV Setup ===
csv_filename = os.path.join(directory, f"sensor_data_{timestamp}.csv")
csv_file = open(csv_filename, mode='w', newline='')
csv_writer = csv.writer(csv_file)
csv_writer.writerow(["x", "y", "temperature", "humidity", "air_quality"])

# === Power CSV Setup ===
power_csv_filename = os.path.join(directory, f"power_data_{timestamp}.csv")
power_csv_file = open(power_csv_filename, mode='w', newline='')
power_csv_writer = csv.writer(power_csv_file)
power_csv_writer.writerow(["timestamp", "SoC", "voltage", "current", "temperature"])

# === Save Functions ===
def save_to_csv(x, y, temp, hum, air_quality):
    csv_writer.writerow([x, y, temp, hum, air_quality])
    csv_file.flush()

def save_to_csv_power(ts, SOC, voltage, current, temperature):
    power_csv_writer.writerow([ts, SOC, voltage, current, temperature])
    power_csv_file.flush()

# === Manual Save Thread ===
def manual_save_command():
    while True:
        command = input("[Command] Type 'save' to export plot image:\n ").strip().lower()
        if command == "save":
            save_requested.set()

# === Socket Thread ===
def handle_socket_connection():
    global temperature_map, humidity_map, air_quality_map
    with socket.socket(socket.AF_INET, socket.SOCK_STREAM) as s:
        s.setsockopt(socket.SOL_SOCKET, socket.SO_REUSEADDR, 1)
        s.bind((host, port))
        s.listen(1)
        print(f"[Socket] Listening on {host}:{port}")

        conn, addr = s.accept()
        print(f"[Socket] Connected by {addr}")
        with conn:
            buffer = ""
            while True:
                data = conn.recv(1024)
                if not data:
                    break
                buffer += data.decode()

                while "\n" in buffer:
                    line, buffer = buffer.split("\n", 1)
                    try:
                        parts = line.strip().split(",")
                        tag = parts[0]
                        values = list(map(float, parts[1:]))

                        # Sensor data: x, y, temperature, humidity, air_quality
                        if tag == "T" and len(values) == 5:
                            x, y, temp, hum, aq = values
                            x = int(x)
                            y = int(y)
```

```
                            if 0 <= x < map_size and 0 <= y < map_size:
                                with lock:
                                    temperature_map[y][x] = temp
                                    humidity_map[y][x] = hum
                                    air_quality_map[y][x] = aq
                                save_to_csv(x, y, temp, hum, aq)
                                print(f"[Data] ({x},{y}): Temp={temp:.2f}°C, Hum={hum:.2f}%,
                                    AQ={aq:.2f}")

                        # Power data: timestamp, soc, voltage, current, temp
                        elif tag == "P" and len(values) == 5:
                            timestamp, soc, voltage, current, temp = values
                            save_to_csv_power(timestamp, soc, voltage, current, temp)
                            print(f"[Power] Timestamp={timestamp:.2f}ms SoC={soc:.2f}%, V={
                                voltage:.2f}V, I={current:.2f}A, Temp={temp:.2f}°C")

                        else:
                            print("[Error] Unexpected number of values:", line)

                    except ValueError:
                        print("[Error] Invalid line:", line)

# === Start Threads ===
threading.Thread(target=manual_save_command, daemon=True).start()
threading.Thread(target=handle_socket_connection, daemon=True).start()

# === GUI Setup ===
plt.ion()
fig, (temp_ax, hum_ax, air_ax) = plt.subplots(1, 3, figsize=(15, 5))

temp_img = temp_ax.imshow(np.zeros((map_size, map_size)), cmap='hot_r', origin='lower', vmin
    =0, vmax=40)
hum_img = hum_ax.imshow(np.zeros((map_size, map_size)), cmap='Blues', origin='lower', vmin=0,
     vmax=100)
air_img = air_ax.imshow(np.zeros((map_size, map_size)), cmap='Greens', origin='lower', vmin
    =0, vmax=500)

fig.colorbar(temp_img, ax=temp_ax)
fig.colorbar(hum_img, ax=hum_ax)
fig.colorbar(air_img, ax=air_ax)

temp_texts = [[temp_ax.text(x, y, "", ha="center", va="center", fontsize=8) for x in range(
    map_size)] for y in range(map_size)]
hum_texts = [[hum_ax.text(x, y, "", ha="center", va="center", fontsize=8) for x in range(
    map_size)] for y in range(map_size)]
air_texts = [[air_ax.text(x, y, "", ha="center", va="center", fontsize=8) for x in range(
    map_size)] for y in range(map_size)]

# === Live GUI Update Loop ===
try:
    while True:
        with lock:
            temp_data = np.array([[cell if cell is not None else np.nan for cell in row] for
                row in temperature_map])
            hum_data = np.array([[cell if cell is not None else np.nan for cell in row] for
                row in humidity_map])
            air_data = np.array([[cell if cell is not None else np.nan for cell in row] for
                row in air_quality_map])

            temp_img.set_data(temp_data)
            hum_img.set_data(hum_data)
            air_img.set_data(air_data)

            for y in range(map_size):
                for x in range(map_size):
                    t = temp_data[y][x]
                    h = hum_data[y][x]
                    a = air_data[y][x]

                    temp_texts[y][x].set_text(f"{t:.1f}" if not np.isnan(t) else "")
                    hum_texts[y][x].set_text(f"{h:.1f}" if not np.isnan(h) else "")
```

```
                    air_texts[y][x].set_text(f"{a:.1f}" if not np.isnan(a) else "")

            temp_ax.set_title("Temperature (°C)")
            hum_ax.set_title("Humidity (%)")
            air_ax.set_title("Air Quality (AQI)")

            for ax in (temp_ax, hum_ax, air_ax):
                ax.set_xlabel("X")
                ax.set_ylabel("Y")

            if save_requested.is_set():
                image_filename = os.path.join(directory, f"sensor_map_{datetime.now().
                    strftime('%Y%m%d_%H%M%S')}.png")
                fig.savefig(image_filename)
                print(f"[Save] Image saved as '{image_filename}'")
                print(f"[Info] Data is being logged in '{csv_filename}'")
                save_requested.clear()

            plt.pause(1)

except KeyboardInterrupt:
    print("\n[Exit] Interrupted by user.")

finally:
    image_filename = os.path.join(directory, f"sensor_map_{datetime.now().strftime('%Y%m%d_%H
        %M%S')}.png")
    fig.savefig(image_filename)
    print(f"[Save] Final image saved as '{image_filename}'")
    print(f"[Info] Final sensor data saved in '{csv_filename}'")

    plt.ioff()
    plt.close()
    csv_file.close()
    power_csv_file.close()
    print(f"[Info] Final power data saved in '{power_csv_filename}'")
```

B

Task Division

Table B.1: Distribution of the workload

	Task	Student Name(s)
	Preface	Weikai Chen
	Nonenclature	Weikai Chen
	Summary	Guangran Ran
Chapter 1	Introduction	Guangran Ran
Chapter 2	Literature study	Guangran Ran
Chapter 3	Program of Requirements	Guangran Ran & Weikai Chen
Chapter 4.1	System Architecture Overview	Guangran Ran & Weikai Chen
Chapter 4.2	Modular Communication via CAN Bus	Guangran Ran
Chapter 4.3	Wireless Communication Design	Guangran Ran & Weikai Chen
Chapter 4.4	Sensor Integration and Environmental Monitoring	Weikai Chen
Chapter 4.5	Ultrasonic Sensing and Motor Control	Guangran Ran
Chapter 4.6	2D Grid Mapping	Guangran Ran & Weikai Chen
Chapter 4.7	Data Storage and Transmission	Weikai Chen
Chapter 5.1	Communication Module Implementation	Guangran Ran
Chapter 5.2	Software Implementation	Weikai Chen & Guangran Ran
Chapter 5.3	Sensing Module Implementation	Weikai Chen
Chapter 5.4	Hardware Implementation	Weikai Chen & Guangran Ran
Chapter 6	Discussion	Weikai Chen
Chapter 7	Conclusion	Guangran Ran
	Arduino code	Guangran Ran & Weikai Chen
	Python code	Guangran Ran & Weikai Chen
	CAD and Figures	Weikai Chen & Guangran Ran
	Document Design and Layout	Weikai Chen & Guangran Ran