Smart IoT Platform

ABY IBERKLEID, Brandeis University, USA
JAMES PETULLO, Brandeis University, USA
(JEFF) PANG LIU, Brandeis University, USA
SHUBHAM KAUSHIK, Brandeis University, USA

We present the design and implementation of a modular and scalable Internet of Things (IoT) platform[1] to enable real-time monitoring, control, and automation across heterogeneous embedded devices. The system supports diverse sensors, actuators, and hybrid sensor-controller modules, interconnected through a hub-and-spoke architecture using the MQTT protocol. A Raspberry Pi-based hub is the central node, managing bidirectional communication between edge devices and a cloud-based API. This infrastructure enables remote device configuration, real-time telemetry collection, and automated rule-based control via an intuitive web interface. The platform integrates various devices, including motion sensors, environmental detectors, cameras, robotic arms, interactive displays, and smart gloves. Each device supports configurable parameters, such as update intervals and gesture-action mappings, enhancing adaptability and responsiveness. The platform's key features include support for modular device integration, dynamic configuration, and edge intelligence through AI-assisted optimization. Experimental deployment demonstrates the system's applicability in smart manufacturing and human-machine interaction scenarios. The results validate the platform's reliability, extensibility, and effectiveness in managing complex IoT environments.

ACM Reference Format:
Aby Iberkleid, James Petullo, (Jeff) Pang Liu, and Shubham Kaushik. 2018. Smart IoT Platform. *Proc. ACM Meas. Anal. Comput. Syst.* 37, 4, Article 111 (August 2018), 10 pages. https://doi.org/XXXXXXX.XXXXXXX

1 Project Description

We have developed a scalable Internet of Things (IoT) platform that enables users to integrate custom embedded modules for capturing, storing, analyzing, and acting on real-time data. Through an intuitive web interface, the platform provides capabilities for seamless data visualization, dynamic device configuration, and user-defined automation rules, such as triggering alerts or controlling actuators when specific conditions are met.

Modular Device Integration. Our system supports a heterogeneous network of embedded system devices, including sensors, actuators, and hybrid sensor-controller modules. A wide range of modules has been integrated:

- **Sensors:** Motion detectors, humidity and temperature sensors, random value generators, cameras, and buttons.
- **Actuators:** Stoplights, LCD screens, fans, buzzers, and a robotic arm capable of executing predefined motions.
- **Hybrid Devices:** Smart gloves and interactive buttons combining gesture recognition and control signaling.

[1]Source code available at: https://github.com/aiberk/embedded-systems-platform

Authors' Contact Information: Aby Iberkleid, aiberkleid@brandeis.edu, Brandeis University, Waltham, MA, USA; James Petullo, jamespetullo@brandeis.edu, Brandeis University, Waltham, MA, USA, larst@affiliation.org; (Jeff) Pang Liu, pangliu@brandeis.edu, Brandeis University, Waltham, MA, USA; Shubham Kaushik, kaushiks@brandeis.edu, Brandeis University, Waltham, MA, USA.

ACM 2476-1249/2018/8-ART111
https://doi.org/XXXXXXX.XXXXXXX

Each device supports dynamic configuration such as update intervals, gesture-to-action mappings, and communication parameters. These settings can be modified remotely, providing high adaptability to diverse industrial and experimental use cases.

System Architecture. The platform follows a hub-and-spoke, event-driven architecture composed of distributed microservices. A Raspberry Pi is the central hub, running essential services such as an MQTT broker, backend server, and local data store. Edge devices communicate with the hub over a local network, using the MQTT protocol for bidirectional messaging.

Embedded Device Framework. Each embedded device in our ecosystem serves two core functions:

(1) **Sensing/Actuating:** Devices perform their tasks—capturing sensor data or executing commands.
(2) **Bidirectional Communication:** Devices maintain real-time connectivity using MQTT, allowing for data reporting and remote reconfiguration. Parameters such as Wi-Fi credentials or operational thresholds can be updated via persistent storage. For example, MicroPython devices can alter network settings through `machine.nvs`, modifying configuration files like `wpa_supplicant.conf`.

Automation and Intelligence. The platform supports user-defined rule creation for automation based on sensor inputs or system states. Furthermore, telemetry data can be analyzed by an integrated AI/ML module to enhance operational efficiency through predictive tuning and adaptive control.

Key Features. The platform emphasizes:

- **Modularity:** A flexible framework for integrating new devices and features.
- **Scalability:** Designed to manage a growing ecosystem of sensors and actuators.
- **Edge Intelligence:** Real-time responsiveness and AI-assisted optimization for local and cloud-based operations.

This project demonstrates a robust, adaptable IoT infrastructure suitable for applications in smart manufacturing, human-machine interaction, and intelligent industrial automation.

2 Hardware Inventory and System Schematic

The initial phase of our project successfully demonstrated a proof-of-concept API capable of communicating with embedded devices over MQTT. This prototype involved an ESP32 device simulating a sensor by publishing random values at a configurable heartbeat interval. A cURL command could dynamically modify the heartbeat, establishing robust bidirectional communication between the device and the server.

Backend, API, and Frontend Development (Lead: Aby). The backend API is implemented in GoLang, utilizing PostgreSQL for device metadata management and InfluxDB for time-series sensor data. A React-based frontend dashboard supports:

- Device registration and management
- Sending dynamic configuration commands to devices
- Defining conditional logic (if-then-that) for automated actuator responses based on sensor input

Static IPs were replaced with DNS-based discovery to resolve initial networking challenges, and MQTT connectivity was improved using mDNS. Future enhancements include extending the condition builder to support chained logical expressions in Conjunctive Normal Form and integrating Redis for improved database read performance.

Device Updates: The list of simple devices has been updated as follows:

(1) *OLED Screen — Completed.* Supports an array of strings with configurable loop speed and includes a ping function for management and system health diagnostics. (Developed in Arduino C++)
(2) *Stop Light LED — Completed.* Operates using an array of three binary values to control red, yellow, and green lights, and includes a ping function.

Both the OLED Screen and Stop Light LED use a framework that allows for swift platform registration and bidirectional MQTT communication.

System Schematic: Figure 1 illustrates the complete end-to-end architecture of our IoT automation platform. Multiple ESP32-based embedded devices—categorized as sensors or actuators—connect to an MQTT broker using unique channels. These devices continuously publish telemetry data or subscribe to control messages. The backend Go-based API service subscribes to all MQTT topics, storing high-frequency sensor data in InfluxDB and application metadata in PostgreSQL. It also evaluates real-time user-defined conditions and dispatches actuation commands via MQTT when logical triggers are met. A React-based frontend allows users to configure devices, define condition-action rules, and visualize system behavior. This architecture establishes a continuous and autonomous control loop, enabling fully programmable edge-to-cloud automation.

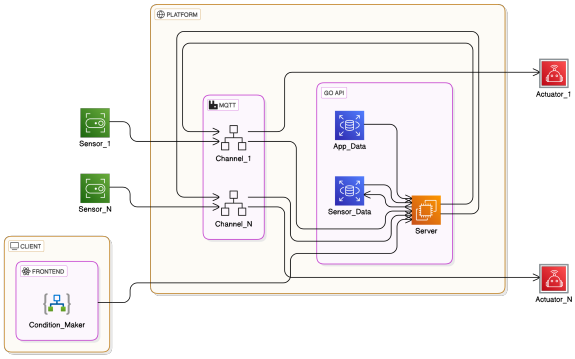


Fig. 1. Backend, API, and Frontend Workflow

Smart Glove (Lead: Jeff). The Smart Glove is a wearable input device developed to interface with our IoT platform. It is designed to interpret human hand gestures in real time and control other connected devices—such as robotic arms—using a combination of ROS and MQTT protocols.

System Summary: The glove integrates two key sensors: the MPU9250 Inertial Measurement Unit (IMU) and the FSR402 force sensor. It is built using Raspberry Pi 4B and Raspberry Pi Pico W, with connectivity and processing managed via a Lenovo laptop. The system uses the Robot Operating System (ROS) to handle sensor data acquisition, gesture recognition, and topic-based communication.

Architecture Overview: The glove communicates through a ROS node that publishes IMU and force sensor data over defined ROS topics. These topics are subscribed to by a motion detector node that classifies the data into gestures such as `UP`, `DOWN`, and `STATIONARY`. The output is then bridged via MQTT to the IoT platform, where the recognized gestures can trigger corresponding actuator responses. The glove can also send commands directly to a robotic arm node, forming a closed-loop gesture-to-action pipeline.

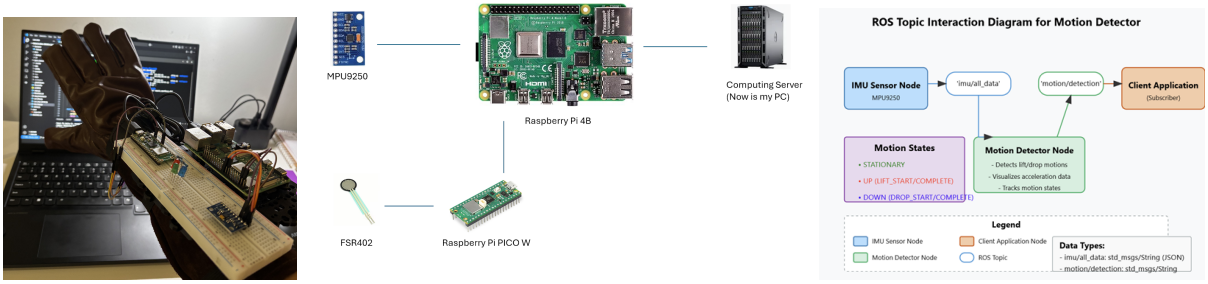


Fig. 2. Hardware and system schematic of the Smart Glove integration with sensors and Raspberry Pi

ROS Topic Pipeline: The system currently supports the following ROS topics:

- `/imu/all_data`: Publishes real-time accelerometer and gyroscope readings.
- `/motion/detection`: Subscribes to IMU data and outputs high-level motion states.
- `/glove/data`: (Planned) Publishes combined IMU and flexible sensor information for MQTT-based IoT control.

Gesture Recognition Logic: The Smart Glove currently recognizes:

- **UP** – Lifting motion
- **DOWN** – Lowering motion
- **STATIONARY** – No motion detected

It also include directional gesture recognition, variable-speed detection, and compound gestures involving force and flex sensors.

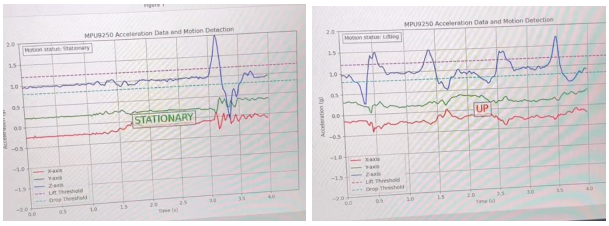


Fig. 3. Recognized gestures: UP, DOWN, STATIONARY

Robot Arm: The robot arm is an output device that can connect to the IoT platform and can be directly controlled by the glove. This robot arm is pre-installed on ESP32, and has integrated into the ROS network. It also uses ROS framework and use a node to communicate with MQTT broker:

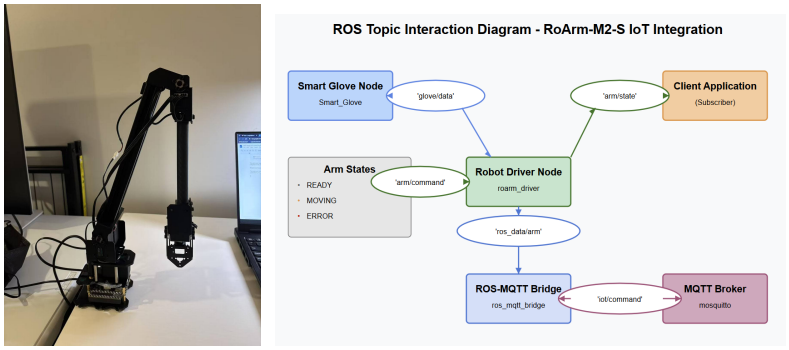


Fig. 4. Robot Arm

The Smart Glove and Robot Arm modules have successfully completed all planned development phases. The glove now supports reliable gesture recognition using IMU and force sensors, and can control external devices—including the robotic arm—through a fully integrated ROS-MQTT communication pipeline. The robot arm, running on ESP32, is seamlessly incorporated into the ROS network and responds to gestures in real time via MQTT commands. This end-to-end interaction enables a rich, intuitive control system that demonstrates the platform's capability for complex, human-centered automation. Together, these components showcase a robust and extensible framework for real-time, sensor-driven IoT systems in industrial and research environments.

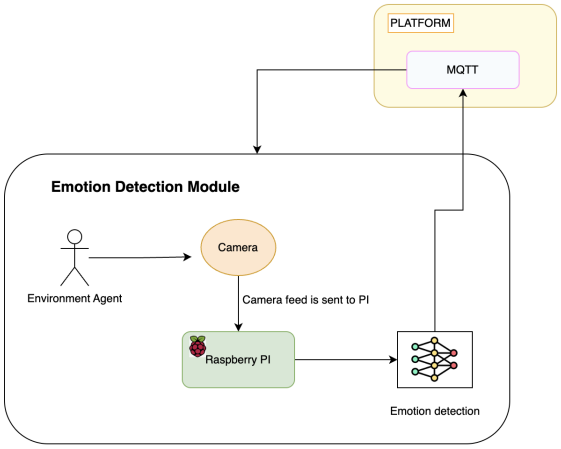
Emotion Recognition (Lead: James). The emotion recognition module is complete and fully integrated with the platform as a whole. The module's hardware includes a **Raspberry Pi 5** with an onboard **Raspberry Pi Camera v2**, connected via a cable. The software consists of a custom API wrapper around the `Picamera2` module, enabling fast and efficient image capture, storage, retrieval, and emotion detection inference. The module interfaces with the platform via an adaptable MQTT event loop, specifically written for usage across devices for this project.

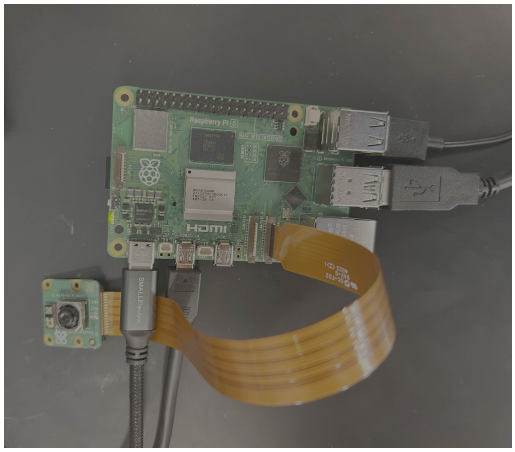
Module Components:

- **Hardware Integration** The Raspberry Pi Camera v2 is securely attached to the Raspberry Pi, and powered by Raspberry Pi OS in conjunction with the `Picamera2` library, the device can capture high-quality images and

store them on the system for use later on. The camera's default settings are preserved, with the exception of the preview window screen size, which is set to be `1000x800`.

- **Image Capture Implementation**: To facilitate image capture, a wrapper API around `Picamera2` was written. This wrapper handles the camera's initialization and stoppage, ensuring that OS errors surrounding unterminated processes are gracefully halted. In particular, the library supports a variety of fallbacks in the event the camera fails to load on a system start. The API stores images in a designated folder (e.g., all_imgs), ensuring that each capture is timed for proper logging, traceability, and retrieval. All files pertaining to the API can be found on our (Github page)
- **Emotion Detection and Analysis**
  Under the hood, the API utilizes the (DeepFace) library to perform emotion detection and analysis. `DeepFace` itself is a wrapper around a number of common facial recognition libraries, including the popular `Yolo` model. Due to its accuracy and speed, we chose `Yolo` as the backend facial recognition model. When emotion detection is to occur, our API captures an image, saves it to the storage folder, and then calls the inference method to retrieve the saved image and perform emotion analysis. In particular, the set of possible emotions that can be registered include $\{angry, happy, fear, surprise, neutral, disgust, sad\}$ The dominant emotion detected in the image is returned by the API for further use across the platform.





- **Platform Integration** The emotion detection module interfaces with the platform via a task event listener. Upon receiving a delegated task, the module captures an imagine, run the emotion detection, and pushes the result back to the platform through the listener client. Other interconnected devices can then utilize the detected emotion for additional tasks (see **Further Avenues of Adaptability** below). For demonstration purposes, we have configured the emotion detection module to work in conjunction with a button and LCD screen. When clicked the button, connected to the platform as a separate device, sends an `fired` event, which is intercepted by the emotion detection module. Then, the emotion read by the module is dispatched to the LCD screen device, were it is displayed for ease of viewing.
- **Further Avenues of Adaptability** For instance:
  - *Lighting Adjustments*: If the system detects that the user appears frustrated or overwhelmed—potentially due to harsh lighting—it could automatically dim the lights to create a more soothing environment.

- *Temperature Regulation*: If a user's expression indicates discomfort from cold conditions, the system could increase the room temperature slightly, ensuring a cozy ambiance.
- *Ambient Sound and Music*: The system might also adjust the background music volume or select calming music if stress or anxiety is detected.
- *Personalized Workspace Settings*: In an office environment, the system could adjust desk lighting and screen brightness or even notify the user to take a break if prolonged stress is detected.
- *Wellness Monitoring*: In healthcare or wellness applications, emotion detection could trigger alerts for caregivers if a patient exhibits signs of distress or significant mood changes.

Mini-Modules and Integration (Lead: Shubham). This module demonstrates a real-time embedded system built around MQTT communication and multi-device interaction. It integrates four components—a push button (input), and three output devices: a fan (simulated using an LED), a buzzer, and an LCD display. All devices are connected to a Raspberry Pi via a breadboard and operate asynchronously through topic-based MQTT messaging.
System Overview: Each hardware module is assigned a unique `device_id` and communicates via MQTT topics like `sensors/<device_id>/data` and `devices/<device_id>/config`. The system uses a local MQTT broker discovered via mDNS (or a fallback IP), allowing for device-agnostic communication. Each modular device runs a custom handler listening to its subscribed topic. (Github page)

Device Description and Schematic Diagram:

- *Button Module (GPIO 17):* When pressed, publishes a JSON payload indicating a click event.
- *Fan (LED on GPIO 22):* Turns ON or OFF based on received messages containing a "fan" key.
- *Buzzer (PWM on GPIO 18):* Plays a melody using frequency values specified in the MQTT payload.
- *LCD Display (I2C via GPIO 2/3):* Updates its two-line display with emotion strings or event messages (e.g., "Fire Detected", "Button Pressed").

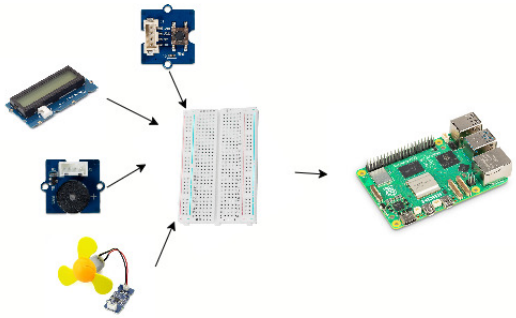


Fig. 5. Fully integrated setup: Button input and Fan (LED), Buzzer, and LCD as output devices—all MQTT-driven and connected through a breadboard to the Raspberry Pi.

Example Use Cases:

(1) *Button Press Trigger (Normal Event):* A user presses the button. The button publishes:

```
{"data": {"isClicked": true, }}
```

The LCD displays `"Emotion: Clicked"`, the fan turns ON, and the buzzer plays a quick tone.

(2) *Fire Alert Scenario:* An external system (e.g., temperature/fire sensor) publishes the following:

```
{"data": {"emotion": "FIRE DETECTED", }}
{"data": {"notes": [988, 988, 988, 784, 784], }}
{"data": {"fan": "true", }}
```

The LCD flashes "FIRE DETECTED", the fan activates, and the buzzer repeatedly plays a high-pitched warning tone.

(3) *Emotion-Based Display:* From a remote interface or controller:

```
{"data": {"emotion": "Happy", }}
```

The LCD shows:

```
Emotion:
 Happy
```

(4) *Manual Fan Override:* MQTT message:

```
{"data": {"fan": "false", }}
```

The fan (LED) turns off even if the button was recently pressed.

Integration Highlights:: Devices are loosely coupled, improving scalability and fault isolation. The 'main.py' script initializes all device clients and sets up handlers in a non-blocking loop using 'loop_start()'. GPIO pins are mapped in a central 'gpio.py' file, and the LCD uses the Grove JHD1802 I2C library for clear abstraction. The 'core/broker.py' file resolves broker IP using mDNS ('platform.local') with a fallback for local deployment.

3 Team Contributions and Challenges

Aby Iberkleid Szainrok

Contributions

- Branding & Design: Developed the platform's visual identity including logo, typography, iconography, and color system. Designed UI mockups, architecture diagrams, and polished final presentation visuals.
- System Architecture: Designed the end-to-end data flow, communication model, and role taxonomy for device types (Sensor, Actuator, Controller).
- API & Backend: Built the core backend in GoLang with PostgreSQL for metadata and InfluxDB for telemetry. Exposed REST/MQTT endpoints to support bidirectional device control and condition-triggered automations.
- Frontend: Developed a React-based interface for device registration, command execution, and a visual rule builder (IF-THEN-THAT).
- Networking & Deployment: Configured a local WLAN router for consistent device connectivity. Deployed all services on Raspberry Pi 4/5 using automated Bash scripts.
- Universal Device Framework: Authored cross-platform C++/Python libraries for ESP32, Raspberry Pi, and MicroPython devices with support for discovery, health checks, and MQTT messaging.
- Firmware Development: Developed multiple device firmwares including:
  - OLED Screen (ESP32, C++): Scrollable text array with dynamic speed control.
  - Stop-Light LED (ESP32, C++): Bit-encoded red-yellow-green status system with diagnostic ping.
  - Randomizer (ESP32, C++): Simulated telemetry sensor for testing.
  - Motion Sensor (Pico W, MicroPython): Live motion data streaming.

Challenges & Solutions

- Unstable HTTP communication: Transitioned to MQTT with mDNS discovery, significantly reducing latency and improving addressability.

- Database load on Raspberry Pi: Adopted a hybrid stack of PostgreSQL and InfluxDB for ARM compatibility and performance under write-heavy conditions.
- Device integration complexity: Designed a minimal payload format that supported a wide range of devices with no need for custom protocol extensions.

Pang (Jeff) Liu

Contributions

- Smart Glove Module: Designed and implemented a wearable glove input system using MPU9250 and FSR402 sensors. The glove interprets physical motion and pressure gestures.
- Hardware Integration: Built the glove using Raspberry Pi 4B and Raspberry Pi Pico W as processing and sensor units.
- ROS Node Development: Created custom ROS nodes to handle sensor publishing and motion detection. Used ‘/imu/all_data‘ and ‘/motion/detection‘ for gesture recognition.
- Gesture Logic: Implemented state detection for gestures like UP, DOWN, and STATIONARY. Planned extensions include speed sensitivity and compound gesture classification.
- ROS–MQTT Bridge: Integrated the ROS framework with the MQTT system to publish gestures and control devices on the IoT platform.
- Robot Arm Control: Developed an ESP32-based robot arm module that receives MQTT commands and performs physical movement in response to glove gestures.
- System Diagrams: Created multiple hardware and ROS interaction schematics to document glove-arm integration.

Challenges & Solutions

- Multi-sensor sync: Carefully synchronized MPU and FSR inputs to prevent false positives.
- Gesture noise: Tuned thresholds and filtering logic in ROS nodes to improve motion classification.
- Realtime feedback: Ensured responsive interaction by optimizing ROS topic frequencies and loop timing.

James Petullo

Contributions

- Camera Integration: Integrated the Raspberry Pi Camera v2 with the Raspberry Pi 5.
- Library Customization: Adapted and configured the Picamera2 library to meet system compatibility needs and performance constraints.
- API Integration: Developed an API wrapper for capturing images, storing metadata, and invoking emotion detection algorithms.
- Emotion Detection: Researched and deployed an optimized emotion detection library suitable for edge inference, enabling devices to respond based on user emotional state.
- Cross-Module Coordination: Assisted teammates in linking the emotion detection service with other IoT subsystems.

Challenges & Solutions

- Hardware constraints: Managed limited memory and processing bandwidth on the Pi 5 by tuning inference parameters.

- Library mismatch: Resolved compatibility issues between Picamera2 and ROS nodes through API adaptation.
- Data flow integration: Developed an intermediary handler for routing emotion data to the central database and UI.

Shubham Kaushik

Contributions

- Mini-Modules Subsystem: Designed and implemented a modular real-time embedded system with one input device (button) and three output devices (fan/LED, buzzer, LCD), all integrated over MQTT.
- Hardware Integration: Built and wired the full system using a Raspberry Pi, breadboard, and GPIO pins to support simultaneous device operation.
- MQTT Communication Logic: Defined message topics and formats for each device. Implemented device-side MQTT clients with unique handlers, supporting asynchronous and decoupled behavior.
- Device Firmware: Wrote Python handlers for:
  - Button Module: Publishes JSON payloads on click events.
  - Fan Module (LED): Turns on/off based on command messages.
  - Buzzer Module: Plays custom tones or melodies defined in message payloads.
  - LCD Module: Displays event-based messages and emotions using a two-line interface over I2C.
- Use Case Development: Defined and tested complex scenarios like fire alerts, emotional responses, and manual device overrides using structured MQTT payloads.
- Code Organization and Documentation: Structured the mini-modules repository with ‘main.py‘, ‘gpio.py‘, and ‘broker.py‘ files for clean initialization, GPIO mapping, and broker resolution using mDNS. Maintained public codebase on GitHub for reproducibility.

Challenges & Solutions

- Asynchronous Device Handling: Multiple devices needed to operate independently without blocking execution. Used MQTT’s asynchronous ‘loop_start()‘ in combination with modular handlers to support concurrent device responses.
- LCD Display Over I2C: Interfacing the LCD over I2C had timing and driver issues. Adopted the Grove JHD1802 library and optimized message formatting to ensure stability and legibility.
- Dynamic Message Parsing: Varying message formats across devices (e.g., notes for buzzer, emotion strings for LCD) introduced parsing challenges. Implemented a flexible payload handler with validation logic per device to support safe operation.
- Broker Discovery: Hardcoded broker IPs hindered portability across networks. Integrated mDNS (‘platform.local‘) with fallback IP resolution to ensure robust connectivity across all deployments.

4 Videos

(1) Smart IoT Platform
(2) Robot Arm
(3) Glove

References

[1] Instructables. *ESP32-CAM Web Server and Getting Started Guide*. Available at: http://instructables.com/ESP32-CAM-WEB-Server-and-Getting-Started-Guide/.
[2] Alex Xu. *System Design Interview – An Insider's Guide*. Independently Published, 2020.
[3] Espressif Systems. *ESP-IDF Programming Guide for ESP32*. Available at: https://docs.espressif.com/projects/esp-idf/en/stable/esp32/index.html.
[4] Serhat Küçükdermenci. *Sign Language Voice Converter Design Using Raspberry Pi for Impaired Individuals*. ResearchGate, 2023. Available at: https://www.researchgate.net/publication/373690087_Sign_language_voice_convertor_design_using_Raspberry_pi_for_impaired_individuals.
[5] Simone Salerno. *Smart Glove with Flex Sensors*. Edge Impulse Documentation, 2023. Available at: https://docs.edgeimpulse.com/experts/novel-sensor-projects/flex-sensors-hci.
[6] Ryan Lowe, Yi Wu, Aviv Tamar, Jean Harb, Pieter Abbeel, and Igor Mordatch. *Multiagent Cooperation and Competition with Deep Reinforcement Learning*. PLOS ONE, Vol. 12, No. 4, 2017. Available at: https://journals.plos.org/plosone/article?id=10.1371/journal.pone.0172395.
[7] ROS Documentation. *ROS 2 Documentation – ROS Humble*. Available at: https://docs.ros.org/en/humble/.
[8] TDK Invensense. *MPU9250 Datasheet*. Available at: https://invensense.tdk.com/products/motion-tracking/9-axis/mpu-9250/.