Michael Montanaro

ECE Capstone Final Report

Software Ecosystem: Design and Implementation

The development stack combines the versatility and speed of Flutter, a cross-platform framework, with the robustness of MQTT, a lightweight messaging protocol, to enhance mobile application functionality. The integration of these technologies orchestrates a seamless and responsive communication infrastructure between the mobile application and two ESP32 microcontroller units housed within the enclosure mentioned above.

1 App Development

The primary framework for application development was driven mainly by the need for cross-platform development. This shortened the playing field a bit, with two competitors standing out amongst the rest: React Native and Flutter. Though React development is well documented thanks to the immense community behind the framework, Flutter holds an advantage in both performance and development time. “Ahead of Time” (AoT) compilation and the use of native languages rather than compiling from Javascript within Flutter gives it the edge in performance. Additionally, development time is also known to be faster on Flutter thanks to the library of widgets that give the developer access to most tools necessary for creating a user interface. The built-in tools along with Flutter’s hot reload feature, which allows the screen to reload without relaunching the entire application, greatly increased the development time of this application. Lastly, because this was done by a single developer, the single codebase for multi-platform development helped to maintain order within the repository, better manage debugging, and optimize resources as no additional time would be needed to develop an application for a different platform (ie. web, android, macOS).

With the decision to use Flutter, came the adoption of Dart as the backbone of this project. Not only were we new to Flutter as a development tool, but no member of the team had any experience in Dart either. However, Dart shares similar syntax to C++ with a few added features by Google, so syntax and a few basic programming projects sufficed for the learning phase for Dart. Next, navigating the Flutter framework posed its challenges, with the framework's swift evolution causing online tutorials and even official documentation to become outdated within a short span of six months. To gain familiarity with the toolkit, the example “Hello World” app was used along with a Dart/Flutter LSP to trial-and-error test some features while reading through the beginning documentation. After achieving basic comfort building a screen, the objective became developing a basic app with the Google Maps Flutter API integrated within the home screen. The API produces a map, which is crucial to the final application design. Shockingly, this, or more specifically integrating the map along with a user-location feature, proved to be a multiple-week endeavor.

The projects mentioned above all served to show the capabilities of Flutter and to develop a user interface. Once the difficulty of various designs became more obvious, the final app could be more comfortably designed to accommodate a timeline. Each element of an app in Flutter is known as a screen, three of which were made for this project: map, station, and list_stations. The resulting design can be seen in the figure below.

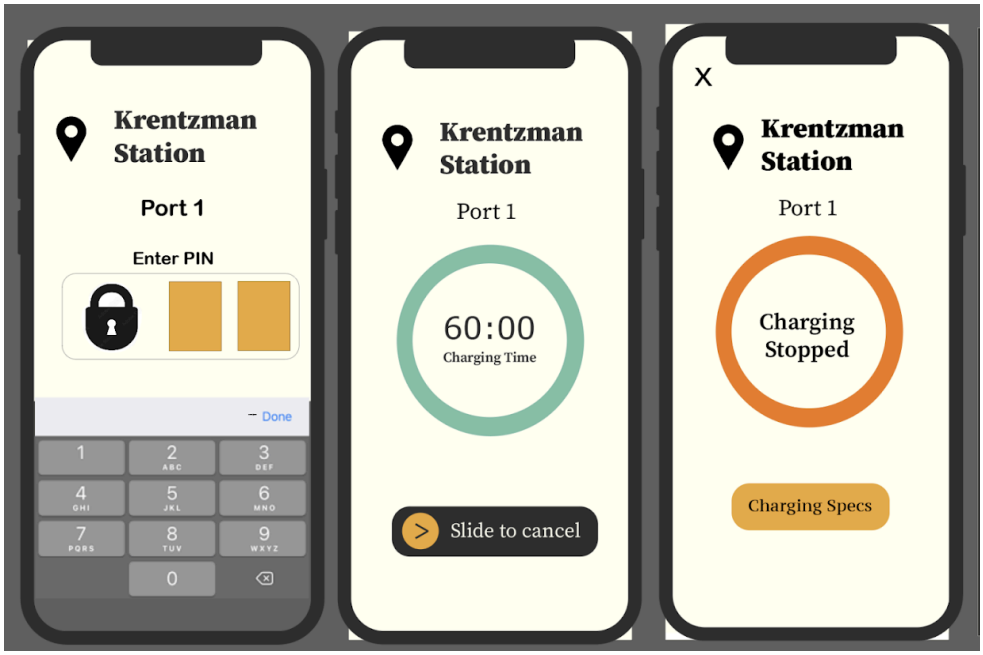

Figure 2.4.1: Final UI design

The development phase commenced with the creation of data models to represent the underlying structure of the application. Models were created for map, to initialize the pin locations for each station's location on the map, and for station, to more easily access metadata about each station such as ID and number of ports. Following this, the screens were implemented, adhering to the predefined UI layouts. The process was relatively smooth, with minor inconveniences arising on the locally downloaded version of the application in preparation for demo day; however, the errors were quickly fixed and the UI was updated to be more dynamic based on screen size. The last step was the integration of MQTT, which included creating a secure connection with an MQTT broker, more on this later, to send and receive messages to and from specified topics and subtopics.

The intentional structuring of the repository and thoughtful design of UI layouts, combined with the efficient execution of data models, screens, and MQTT integration, produced a cohesive and functional application. Many thanks to the countless GitHub tutorial repositories that helped guide me through the structure of Flutter applications, which proved essential in calming the chaos that is Flutter repositories.

2 Microcontroller Software Development

The decision to employ ESP32 microcontrollers for this project was primarily propelled by their versatility and notably low power consumption, aligning seamlessly with the project's IoT requirements. Leveraging ESP32's built-in Wi-Fi capabilities, robust processing power, and support for various communication protocols, the microcontrollers proved to be an ideal choice.

In the software development phase, the first ESP32 was designed for effective charge state management using a software-based Pulse Width Modulation (sPWM) waveform. The primary goal was to optimize the control of output charge from an outlet connected to our electrical enclosure, specifically for charging e-mobility devices. The sPWM algorithm, crafted for precise waveform generation, offered a versatile method for toggling charge states while ensuring efficient and controlled power delivery.

Simultaneously, the second ESP32 was tailored to generate secure Two-Factor Authentication (2FA) codes, aligning with the project's commitment to robust security. Specialized algorithms were implemented to ensure dynamic and secure code generation. The integration of a seven-segment display served as a visually accessible output for the generated codes.

Both ESP32 units were designed for MQTT communication to relay their charging states and generated 2FA codes, as well as to receive requests from the user-application to enable/disable charge and to generate the 2FA code. This design choice not only streamlined communication but also reflected a holistic approach to efficient charge management.

3 Messaging Protocol

The communication protocol between the microcontrollers and the user application was initially intended to be facilitated through AWS IoT Core, utilizing the MQTT protocol for seamless message exchange with the IoT MQTT Broker. However, recognizing the potential increase in both cost and development time associated with AWS IoT Core, an alternative solution was sought to maintain efficiency without compromising functionality.

As a strategic decision, we opted for a customized setup using Eclipse Mosquitto MQTT broker, hosted within a Docker container on a secured EC2 instance. Leveraging the free tier version of AWS EC2 instance, this solution provided a cost-effective infrastructure for the project. The EC2 instance was rigorously locked down, permitting communication solely through port 1883. Within this secured environment, port forwarding was configured to redirect the Docker container's port 1883 to the corresponding section on the EC2 instance. Access to the MQTT broker residing in the Docker container was regulated by requiring initiation through a specified username and password system. This approach not only capitalized on the cost savings offered

by the AWS free tier but also allowed for a tailored and secure communication infrastructure. The combination of the microcontrollers and the user application proved seamless, underscoring the inherent simplicity of the MQTT protocol, which facilitated efficient communication between the components.

4 Development Process

The development process for this project followed a structured and collaborative approach, utilizing version control through GitHub to manage code repositories for the ESP32 microcontrollers, Flutter application, and simulation code generated from Simulink/MATLAB. Each component's codebase was housed in its respective repository within the main GitHub repository, ensuring organized version tracking and collaboration among team members. This centralized version control approach facilitated seamless coordination, allowing for efficient monitoring of changes, collaborative contributions, and the ability to revert to previous states if needed.

5 Results

Our system seamlessly integrates the Flutter app with ESP32 microcontrollers, featuring a robust software-based Pulse Width Modulation (sPWM) for precise charge state control. The app serves as a user interface, allowing users to verify Two-Factor Authentication (2FA) codes, connect to a station at a specified port, and enable charging for e-mobility devices such as e-bikes and e-scooters. The user-friendly Flutter interface enhances accessibility, providing clear controls for efficient and straightforward interaction. The images below give a concise visual overview of the final app state, showcasing these key functionalities.

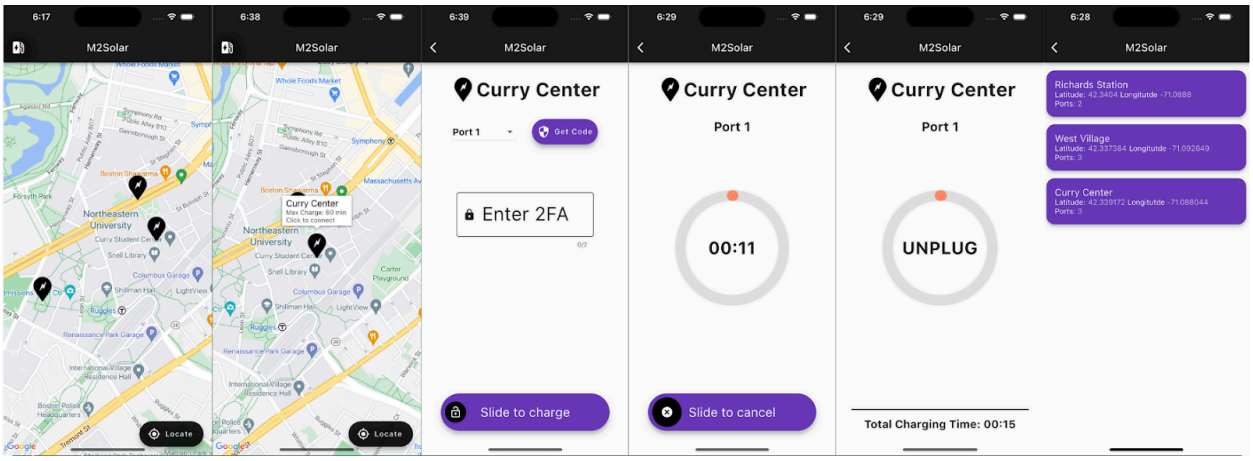

Figure 6.1.4: Final Capstone Application built with Flutter

6 Improvements

6.1 Feature Additions

In the current system, functionality has been implemented both to set the charge state to a specific port on the station, and to provide the capability to receive charge data from the initiated charging cycle. However, due to hardware time constraints, we were unable to fully explore and integrate this feature into the final project. Moving forward, we would like charging data to be accessible to the user along with adding a user login system using Flutter Firebase, which could track total usage within the system. This addition would enhance security, allow for personalized user interactions, and create the platform to add a payment system for a more production-ready product.

Furthermore, to enhance user experience, the introduction of more user notifications could be beneficial. For instance, implementing a notification system to alert users if the seven-segment display is currently in use for another user's Two-Factor Authentication (2FA) code could prevent conflicts and improve the overall reliability of the system. This conflict could also be resolved by expanding the system to incorporate more seven-segment displays, preparing for a larger pool of users. These enhancements, collectively, would contribute to a more refined and versatile product with the potential for widespread adoption.

6.2 Design Improvements **appended**

This section outlines the design enhancements implemented during the independent study conducted in Spring 2024 with Professor Kimani. The focus was on comparing the performance of dual-microcontroller systems. The original solution, described above, comprised two ESP32 microcontrollers communicating with an external server via MQTT. The overhead incurred by calls and filtering through topics introduced issues of time, complexity, and security concerns. Therefore, the server interactions should be limited and done outside of a main software functionality. This would be the sPWM generation used to toggle output ports for the solar panel or battery. The improvement goals were to minimize external connections, reduce complexity, and enhance system performance.

6.2.3 Proposed System

With the above requirements in mind, an optimal approach involves isolating the core functionality within a single microcontroller tasked with listening for state change requests through a wired communication protocol (UART in this case). The centralized microcontroller, a Sipeed Longan Nano, serves as the hub for managing state changes triggered by external messages read from the UART port. To facilitate two-factor authentication and communication

with the external server, the client-side messages are filtered through the second microcontroller (an ESP32), and requested state changes are transmitted using the UART protocol.

The system took two software development kits, one for each of the ESP32 and the Sipeed Longan Nano. The former required use of the Espressif-IDF and the latter used the Nuclei SDK. Both tools come preinstalled with tools to setup common communication protocol available on the boards. These libraries helped to assist in developing the prototype and will help to convert the source code.

6.2.3 System Diagram

Figure 6.2.3.1: The ESP32 Wroom DevKit Full Pinout. This project requires use of 1 GND pin, and one set of UART pins. In the case of this project that is TX0 and RX0.

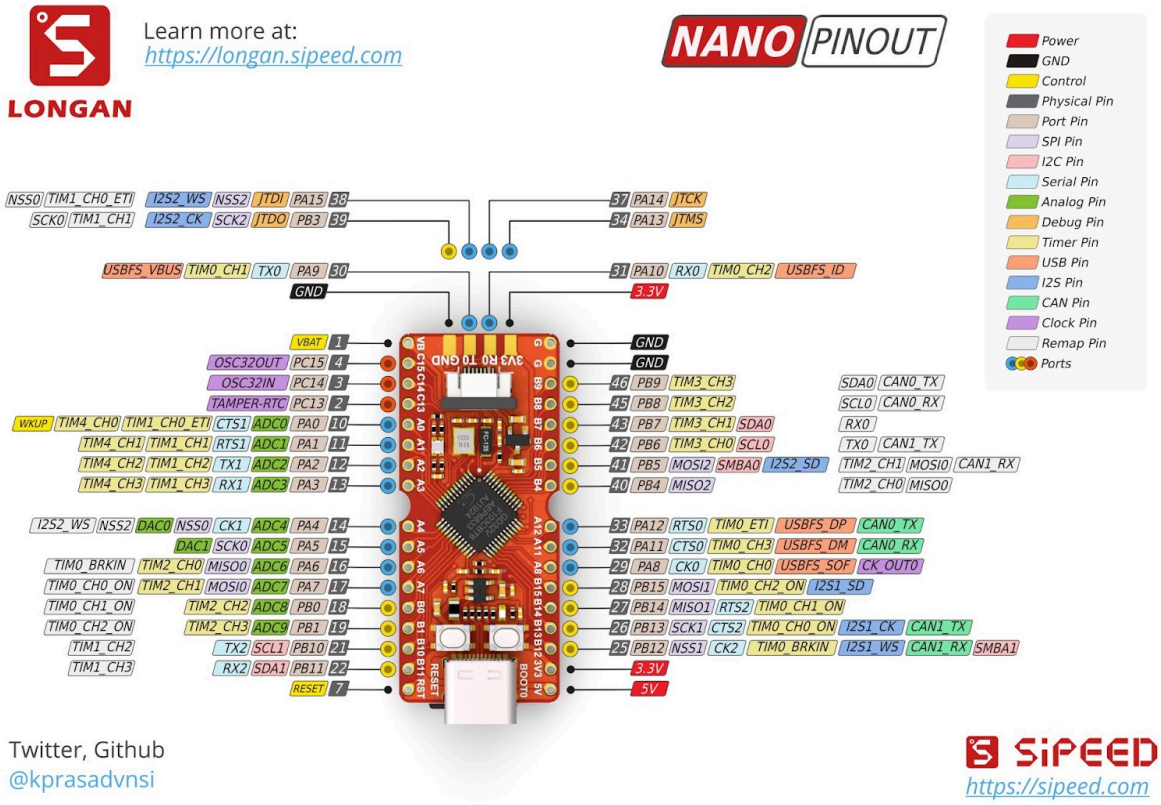

Figure 6.2.3.2: The Sipeed Longan Nano Full Pinout. This project requires use of 1 GND pin, and one set of UART pins. In the case of this project that is TX1 and RX1.

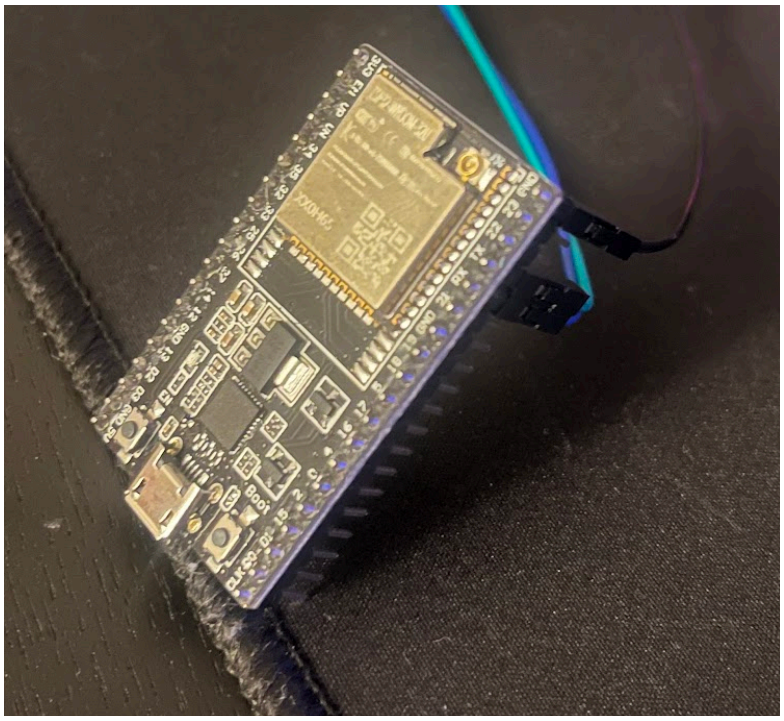

Figure 6.2.3.3: Unpowered ESP32 WROOM wired at TX0, RX0, and GND in prototype.

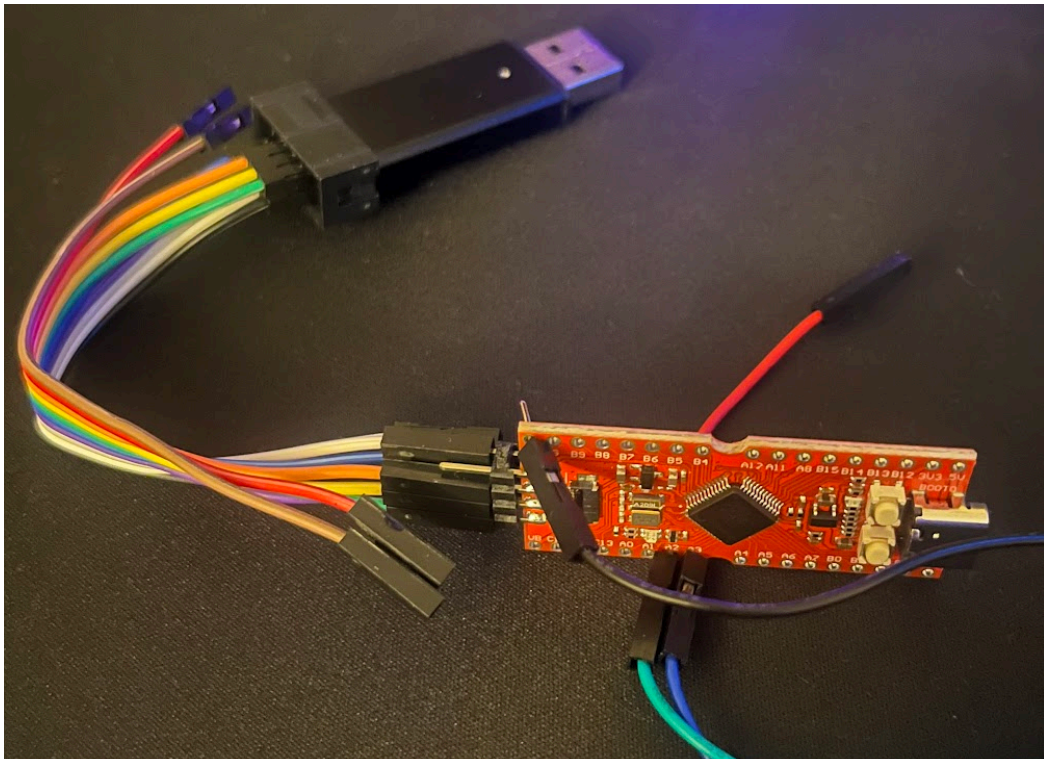

Figure 6.2.3.4: Unpowered SIPEED Longan Nano wired at TX1, RX1, GND in prototype.

6.2.3 Comparison & Analysis

The proposed system exhibits several advantages upon comparison. Firstly, the simplified architecture yields improved performance by consolidating central code onto the Sipeed Longan Nano, a notably more power-efficient controller. Secondly, the wired communication protocol enables real-time control over state changes, leading to improved responsiveness and reliability. Furthermore, this design improves security by isolating core functionality from external server interaction, mitigating potential vulnerabilities and enhancing data integrity. The system enhances scalability and flexibility by decoupling server interactions from main software functionality, facilitating easier modifications and expansions in the future.