Universitatea Politehnica din Timișoara

BATTERY MANAGEMENT SYSTEM FOR SMALL ELECTRIC VEHICLE

Candidate: Robert-Nicolas TRIF

Scientific coordinator: Conf. dr. ing. Răzvan BOGDAN

Session: June 2024

ABSTRACT

Lucrarea prezentată în cadrul proiectului de diplomă se concentrează asupra dezvoltării unui sistem eficient și sigur de încărcare a ansamblului de baterii din interiorul unui vehicul electric. Scopul acestui sistem este să realizeze o încărcare echilibrată a celulelor bateriei, urmărind eliminarea factorilor de risc precum supraîncălzirea și supraîncărcarea. Acești factori pot reduce semnificativ durata de viață a bateriilor, afectând performanța și utilitatea generală a autovehiculelor electrice. În plus, prototipul vehiculului electric este dotat cu metode de siguranță și protecție pentru a asigura o deplasare în condiții optime.

Întregul sistem poate fi împărțit în două categorii distincte: sistemul de încărcare și monitorizare, precum și sistemul de deplasare în condiții de siguranță al vehiculului electric.

Sistemul de încărcare asigură o alimentare eficientă și sigură a bateriilor, utilizând un convertor de tip "step-down" controlat de un microcontroller ESP32 și alimentat de o sursa de tensiune de current continuu. În același timp, sistemul monitorizează constant tensiunea bateriilor și curentul în timpul încărcării cu ajutorul unui senzor de current dedicat. Temperatura bateriilor este monitorizată cu ajutorul a doi senzori de temperatură. De asemenea, sistemul de monitorizare este echipat cu o conexiune către cloud, facilitând analiza în timp real a parametrilor ansamblului de baterii.

Deplasarea în condiții de siguranță este asigurată de către patru motoare electrice, care sunt conectate la un controller de motoare L298N și care funcționează în strânsă legătură cu un senzor ultrasonic, amplasat în partea anterioară a vehiculului electric. Acest sistem permite vehiculului să detecteze și să evite obstacolele în timp real, contribuind la o conducere sigură și fără incidente.

Controlul motoarelor electrice și monitorizarea în timp real se va face cu ajutorul unei aplicații mobile construită pentru sistemul de operare Android. Prin intermediul aplicației, utilizatorul poate primi notificări în momentul în care parametrii bateriilor depășesc anumite limite iar siguranța vehiculului în timpul încărcării poate fi pusă în pericol.

ABSTRACT

The main focus of the project relies on the development of an efficient and safe charging system for the battery pack found inside of an electric vehicle. The purpose of this system is to achieve balanced charging of the battery cells, aiming to eliminate risk factors such as overheating and overcharging. These factors can significantly reduce the life span of the batteries, affecting the performance and the overall utility of electric vehicles. Additionally, the electric vehicle prototype incorporates safety measures in order to optimize driving conditions.

The entire system can be divided into two distinct categories: the charging and monitoring system and the safety driving system of the electric vehicle.

The charging system ensures an efficient and safe power supply to the batteries, using a step-down converter controlled by an ESP32 micro-controller and powered by a DC voltage source. At the same time, the system constantly monitors the battery voltage and current during charging using a dedicated current sensor. The battery temperature is monitored using two temperature sensors. The monitoring system is also equipped with a cloud connection, facilitating real-time analysis of the battery pack parameters.

Safe driving is ensured by four electric motors, which are connected to an L298N motor controller. They operate in close relation with an ultrasonic sensor located at the front of the electric vehicle. This system allows the vehicle to detect and avoid obstacles in real time, contributing to safe and incident-free driving.

Management of the electric motors and real-time monitoring will be accomplished through a dedicated mobile application for the Android platform. This application enables users to promptly receive notifications in case the battery metrics exceed predefined thresholds or if there is a potential compromise to the vehicle's safety during the charging process.

CONTENTS

LIST OF FIGURES

LIST OF TABLES

1. INTRODUCTION

1.1 INTRODUCTION

A battery management system that ensures the safe charging of a pack of 2 NiMH batteries connected in parallel via a buck converter. This battery pack powers an L298N motor driver, which controls 4 electric motors and an ESP32-DevKitC development board, which serves as the control unit for the entire system.

To monitor and visualize the battery parameters, the system includes a mobile application specifically designed for Android users. The application displays relevant information related to the battery's charging process and its specific values and it offers an intuitive interface for controlling the actuators. The connection between the ESP32 and the application is supported by Google Cloud Services, specifically utilizing Firebase's Realtime Database APIs while the notifications are managed using Firebase Functions.

The system has a wide range of features based on the analysis of data collected from the sensors, providing dynamism and a touch of autonomy to the entire process. These features include:

- Monitoring the temperature of both batteries
- Monitoring battery level and voltage
- Monitoring the input voltage value of the power supply used for charging
- Dynamically charging the batteries using the step-down converter
- Ensuring precise movement of the electric vehicle using the motor driver and actuators
- Prevent unwanted collisions using an ultrasonic sensor
- Sending notifications in case of high battery temperature; when the batteries are fully charged or close to 10%
- Providing overcharging protection

1.2 CONTEXT

In today's era of advancing technology and growing environmental consciousness, moving toward sustainable transportation solutions is crucial. Electric vehicles (EVs) offer a promising alternative, significantly reducing emissions and improving energy efficiency. This initiative is very important in reducing pollution caused by internal combustion engine (ICE) vehicles, which heavily contribute to air pollution and greenhouse gas emissions. However, two critical challenges persists when referring to electric vehicles: maximizing the lifespan of batteries and improving the overall autonomy and safety.

Legislative changes targeting stricter reductions of nitrogen oxides (NOx), hydrocarbons (HC), carbon monoxide (CO), and particulate matter (PM) in the exhaust emissions of newly sold vehicles (for example, the EURO VI standard in the European Union and European Economic Area, with EURO VII under development) [1], along with growing awareness among car users, have steered automotive development towards a new direction. This shift places significant emphasis on incorporating low-emission drive systems as a key factor[2]. These restrictions are based on the negative impact of exhaust gases on human health, particularly focusing on respiratory problems caused by inhaling these harmful particles.

Table 1. The seven main categories of air pollutants.[a]

Pollutant	Chemical formula	Cause	Main effect	Magnitude [Mt]
ammonia	NH_3	agriculture	eutrophication	3.9
carbon monoxide	CO	commercial, institutional, and household fuel combustion	global warming	21
nitrogen oxides	NO_x	transport and energy sectors	shortening of life expectancy acidification	7.8
non-methane volatile organic compounds	$C_nH_{2n+2\,(n>1)}$, C_nH_{2n}, $C_nH_{2n+1}Cl$, $C_nH_{2n+1}OH$	industry	precursor to PM and ozone	6.7
ozone	O_3	NMVOCs, NO_x	damage to crops	–
particulate matter[5]	SO_4^{-2}, NO_3^-, NH_4^+, C, C_nH_{2n}, C_nH_{2n+2}, Si, Na^+	transport sector commercial, institutional, and residential	shortening of life expectancy	3.1
sulfur dioxide	SO_2	energy sector	acidification	3.1

[a] Output data estimate of EU28 based on fuel sold in 2014.[6]

Figure 1.1: “Main pollutants together with their cause, effects, and magnitude”[3]

Studies have shown a growing trend in EV sales over the last few years, despite the broader market decline caused by the COVID-19 pandemic. In 2020, while global passenger car sales fell by 16%, electric car sales grew by 43%, resulting in approximately 3 million new electric cars worldwide. This significant increase has created a need for enhanced safety and fire protection measures.[2]

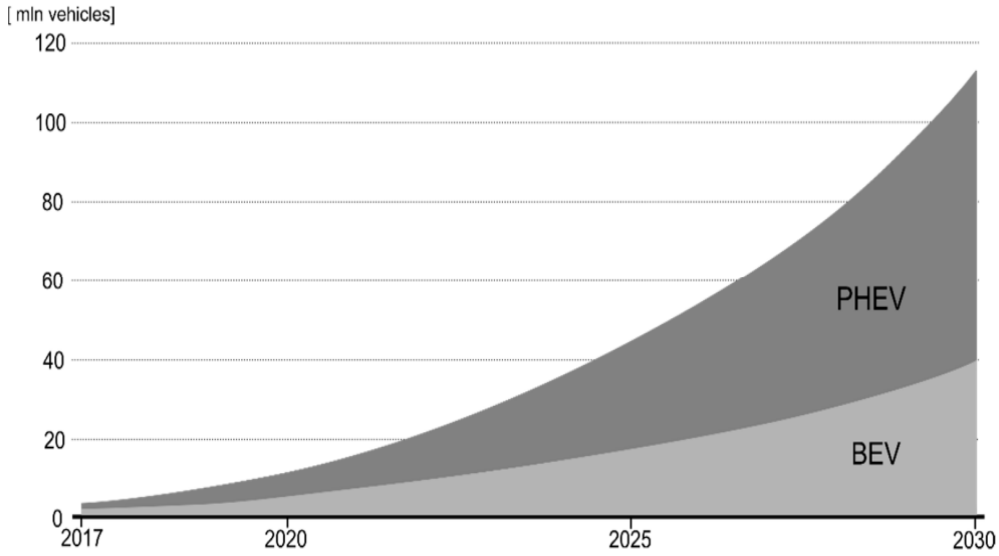

Figure 1.2: “Worldwide car growth forecast by IEA” [4]

As EV adoption continues to rise, addressing the fire hazards associated with these vehicles has become a critical concern. There have been notable incidents of EV fires, indicating the importance of developing potent safety protocols and fire protection strategies.

There were some alarming fires involving Tesla Model S electric cars, some of the first mass-produced EVs. The first incident occurred in October 2013 on a highway in Washington state, where a piece of metal on the road punctured the car’s battery, causing a fire. Two more Tesla fires happened that same month. In Mexico, a driver hit a tree and in Tennessee, a driver ran over a tow bar that damaged the battery compartment. Fortunately, no one was injured in these fires, but they raised concerns about electric car safety.[2]

Figure 1.3: “Worldwide car growth forecast by IEA” [4]

Having in regard the points made by the studies shown below, it is straight-forward the importance of a close monitor and alert system. Essentially, the proposed solution tries to highlight the fact that electric cars are a good alternative for the future. However, both drivers and manufacturers must exercise great care in their use and construction.

1.3 MOTIVATION

As a young teenager, I graduated from the national college and made the decision to pursue my dream of becoming an engineer. I started this journey by enrolling at the University Politehnica of Timisoara where I rediscovered myself and my passions. While my first year of university was not without challenges, particularly due to the pandemic, the following years proved to be an entire adventure.

During this time, I discovered my passion for cars and technology trends and so, my desire was to merge this interest with my enthusiasm for programming and software development.

Over the course of these four years, I dedicated myself in order to learn and grow. I extensively researched topics related to engineering, programming and software development through resources such as YouTube and specialized websites. Simultaneously, I progressed in my programming skills through my university studies. Eventually, my dedication and expertise led to an employment opportunity at Vitesco Technologies, where I was hired to apply my knowledge and skills professionally in the Embedded Systems area.

The idea for my bachelor's project was inspired by my team leader, who noticed my burning desire and attraction for cars and my knowledge of the current literature in this field. Gradually, my focus shifted towards electric vehicles, based on current trends in the automotive industry and my previous experiences related to invention competitions in which I participated in the past.

My motivation was primarily driven by the desire to prove to myself that I can create a complex system that integrates software, electronics, and capabilities related to mobile and embedded software development. I also aimed to develop something practical and tangible, something I could physically interact with and that serves a practical purpose.

One of the biggest challenges I faced while working on this project was adapting to the hardware aspect, specifically the electronics. Initially, I wanted my thesis project to focus on software development but creating a Battery Management System involves much more. Over time, I revisited concepts from the early years of my studies and gained a deeper understanding of them. Working with actual components like sensors, development boards and transistors proved to be quite challenging to understand, but incredibly useful once I understood how they all work together. The entire process was a significant challenge but also a valuable learning experience.

2. STATE OF THE ART

Nowadays, all electric cars have dedicated electronics to monitor the battery's parameters. It's well-known that electric vehicles rely on these components to manage the charging process and prevent battery degradation. This is why monitoring, protecting and optimizing the battery lifespan is crucial for such a modern vehicle.

In addition, most of the car manufacturers prioritize the interaction between the driver and the vehicle, particularly in this case how all the information is presented to the user. It is also important to have a user interface or a "vehicle navigation" system that provides the driver all relevant information about the car (e.g. the battery temperature, battery voltage, charging estimates and more).

Although my project won't offer a complex charging process like those in EVs, it will provide full control over the charging current and voltage. BMSes use specially designed ICs that compute and process various values related to the charging process, making them relatively expensive. As an equivalent for the navigation systems, a mobile application was implemented which aims to closely replicate an information system that monitors battery and charging processes.

Currently, there are two conventional methods of charging batteries using the conductive technique in electric car industry:

- Conventional AC on-board charger shown in Figure 2.1, which focuses on converting AC power from a standard electrical outlet into DC power to charge the battery using the vehicle's on-board charge.

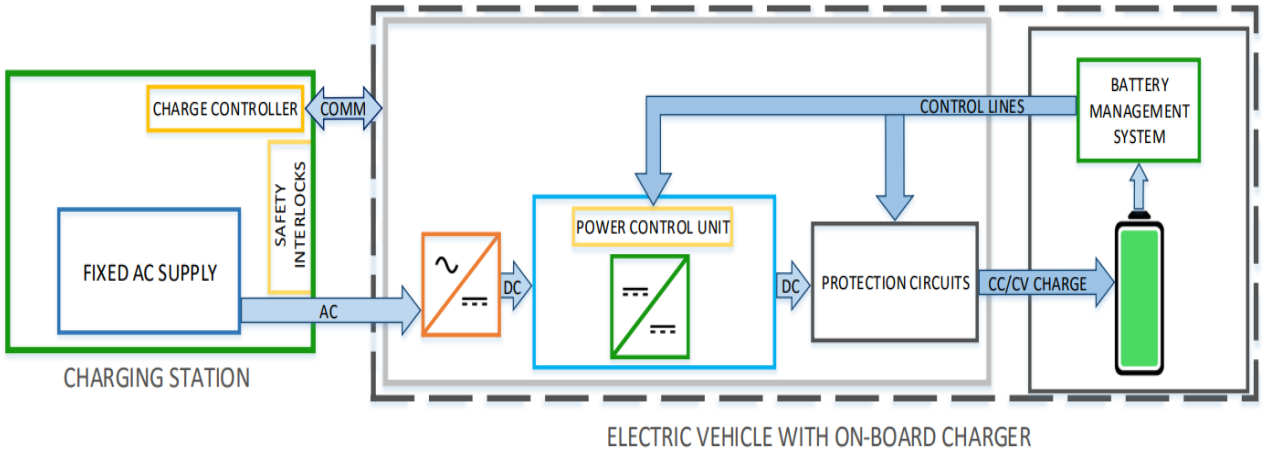

Figure 2.1: "Conventional AC on-board charger configuration"[5]

- Conventional DC off-board charger which directly supplies DC power to the battery from an external charging station, excluding the need for an on-board AC/DC converter and allowing for faster charging times (figure 2.2).

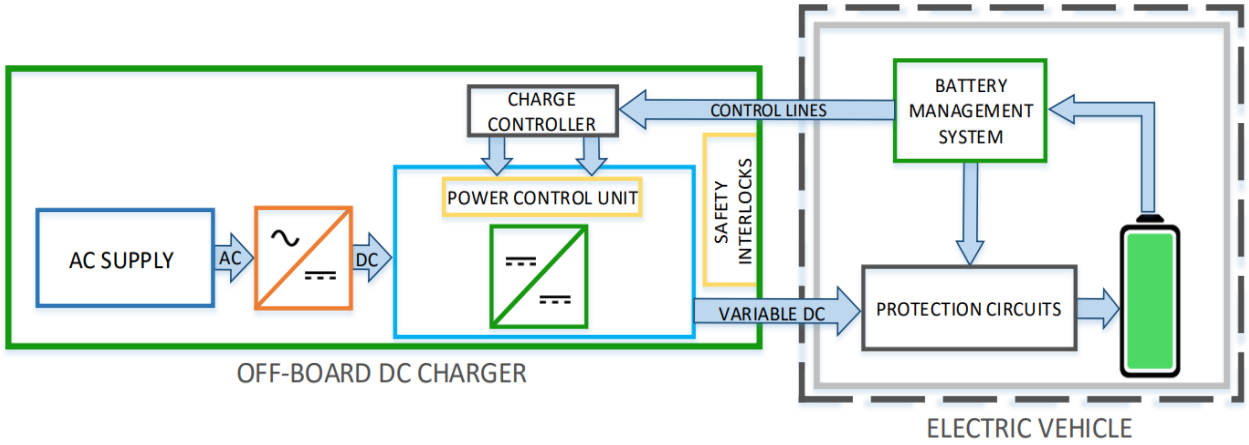

Figure 2.2: ““Conventional DC off-board charger configuration”[5]

Both configurations incorporate a BMS and circuits designed to protect the battery, such as overcharge protection, short-circuit protection and overheat protection. However, this thesis will focus on the on-board charging system. This focus is primarily because electric car drivers frequently opt to charge their batteries using standard home electrical outlets. This method is preferred due to the convenience and cost savings it offers. By using the existing household AC infrastructure, drivers avoid the need to invest in a separate, often expensive, off-board DC charger.

On-board charging systems offer flexibility, enabling users to charge their vehicles at various locations without relying on specialized charging stations. This approach enhances the practicality and accessibility of electric vehicles. Nevertheless, most EVs are designed to support both on-board and off-board charging methods, adapting to various circumstances and ensuring the EV is as reliable and versatile as possible.

Tesla, Inc., Volkswagen AG, BYD Auto Co., Ltd. and Bayerische Motoren Werke AG (BMW) are just a few of the most important competitors that offer a wide range of electric vehicles equipped with built-in charging systems and impressive monitor features.

3. USED TECHNOLOGIES

3.1 HARDWARE

3.1.1 ESP32-DEVKITC

The ESP32-DevKitC is a compact and entry-level development board within the ESP32 series, ideal for a wide range of IoT applications. It offers an extensive set of peripherals and a versatile pin-out for flexible prototyping. The board includes Wi-Fi and Bluetooth capabilities, making it an excellent choice for developing connected electronic projects. Additionally, it supports multiple communication protocols such as UART, SPI and I2C.

It offers a powerful ESP32 240MHz DOWDQ6 micro-controller chip, the DevKitC provides high computational performance. The board also includes essential features such as voltage regulation, programming ICs and 38 GPIO pins (26 configurable pins), facilitating connections to various components.

Among the configurable pins, there are multiple 12-bit analog-to-digital converters (ADCs) for reading and interpreting input voltage values, function that will be presented late in the implementation part [6].

Another important aspect of the ESP32 is its nominal operating voltage of 3.3V. While this can make interfacing with devices requiring different voltage levels a bit challenging, it offers the significant advantage of reduced power consumption.

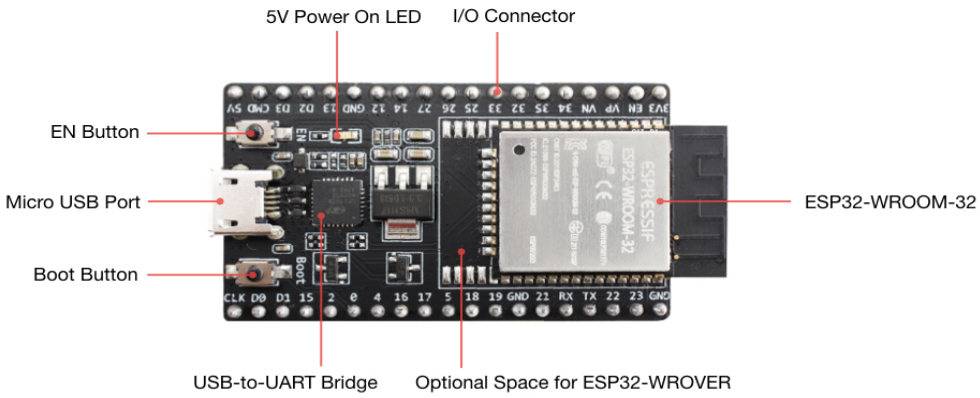

Figure 3.1: “ESP32-DevKitC V4 with ESP32-WROOM-32”[6]

3.1.2 INA219

The INA219 is a high-side current shunt and power monitor equipped with an I2C interface. It can monitor both shunt voltage drop and supply voltage, featuring programmable conversion times and filtering options. With a programmable calibration value and an internal multiplier, the INA219 provides direct reads in amperes. Additionally, it includes a register that multiplies readings to calculate power in watts. The I2C interface supports 16 programmable addresses. [7].

Module's chip is connected to a shunt resistor on the bus of interest and is powered by a supply voltage between +3V and +5.5V. It can monitor bus voltages ranging from 0V to 26V without special power sequencing requirements.

In its normal operating mode, the INA219 continuously converts shunt and bus voltages using its internal analog-to-digital converter, which are then used to calculate current and power. These calculations occur in the background, ensuring efficient operation.

Within this project, the focus will be only on a few specific features to extract relevant insights: load voltage, current and power measurement.

3.1.3 L298N MOTOR DRIVER

The L298N motor driver is a versatile integrated circuit (IC) capable of independently controlling two DC or stepper motors in a standard configuration. This dual H-bridge driver provides two motor channels, allowing it to independently manage the speed and direction of each motor, which enables precise maneuvering of a prototype car.

Operating within a wide voltage range, the L298N typically supports input voltages from 5V to 35V, offering flexibility for different motor types and power requirements. It can handle up to 2A of total current per channel, ensuring sufficient power delivery for most small to medium-sized motors used in car prototypes.

The driver also features built-in protection mechanisms such as over-temperature shutdown and over-current protection, enhancing its reliability in complex applications. Additionally, it includes a 5V channel that can function either as a 5V input or a 5V output, thanks to the built-in voltage regulator, depending on the user's needs [8].

By interfacing with a development board such an ESP32-DevKitC, the L298N motor driver can be easily integrated into the car's control system. This integration enables complex tasks like differential steering and precise vehicle navigation, making the L298N an essential component for developing functional and responsive car prototypes.

3.1.4 US-100 ULTRASONIC SENSOR

The US-100 ultrasonic sensor is a widely used distance measurement device that operates on the principle of echolocation. It emits ultrasonic waves through its transmitter and measures the time taken for the echo to return to its receiver. By calculating the time interval between emission and reception, the sensor accurately determines the distance to objects within its range.

This sensor is compatible with the 3.3V logic level of the ESP32 micro-controller, making it an ideal choice for integration in projects requiring precise distance measurements. It is not only convenient but also very practical, as it eliminates the need for an additional component to convert 5V signals (which is common in most ultrasonic sensors) to 3.3V signals and vice-versa.

In the context of my EV car prototype, the US-100 ultrasonic sensor is particularly suitable as it has been integrated for backward parking assistance. This integration helps in detecting obstacles behind the vehicle, ensuring safer and more efficient parking. The sensor's robust design and ease of use make it a valuable component in the overall functionality of my EV car, enhancing both safety and automation.

3.1.5 NIMH BATTERIES

NiMH batteries or Nickel-Metal Hydride are a type of rechargeable battery that are well-known for their high energy density but essentially for their environmental friendly behavior.

During charging, an electric current oxidizes nickel hydroxide in the positive electrode to nickel oxyhydroxide, while the negative electrode absorbs hydrogen ions, forming a hydride. Upon discharging, the process reverses, providing electric energy. This cycle can be repeated many times, making NiMH batteries highly suitable for applications requiring repeated charging and discharging.

One primary reason for choosing NiMH batteries is their safety. NiMH batteries are safer than lithium-ion batteries because they are less sensible to thermal exposure and do not require protective measures against overcharging and overheating. This makes them simpler to manage in an electronic project environment.

For the second consideration, their compatibility with charging using a step-down converter is an important factor. Properly designed step-down converters can manage the voltage and current to ensure safe charging of NiMH batteries. It is also important to remember that NiMH batteries can tolerate light overcharging because they can transform excess charge into heat. This characteristic is beneficial for a charging system as it can indicate the end of

the charging process.

Thirdly, the evolution of NiMH cells for commercial use highlights their increasing capacity and energy density. From around 1100 mAh in the early 1990s to 2610 mAh by 2005, and projected improvements targeting 3000 mAh, these advancements have made NiMH batteries a preferred choice for consumer electronics, power tools and electric vehicles, where reliable rechargeable power sources are essential [9].

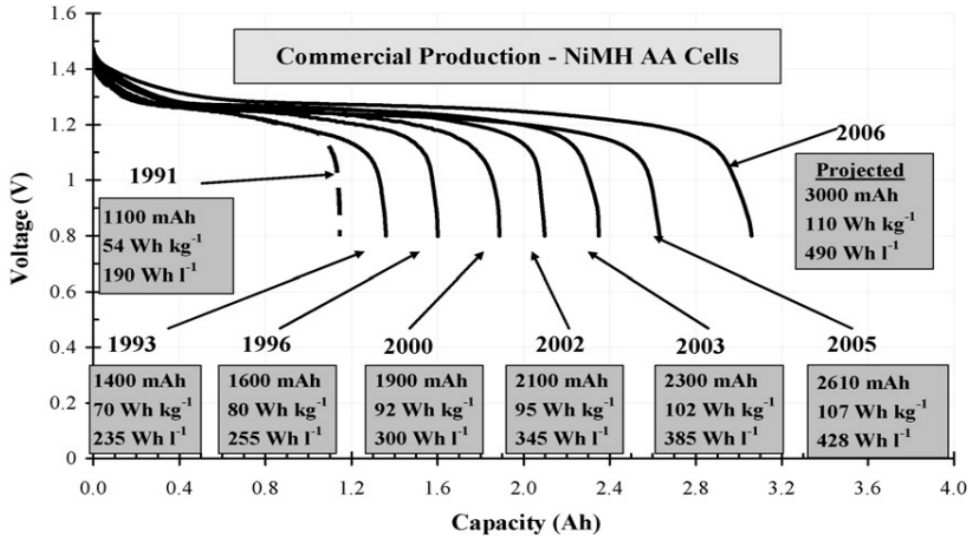

Figure 3.2: “Evolution of commercial NiMH batteries energy”[9]

In my project, there are going to be used two 9V NiMH 200mAh batteries connected in parallel, providing a combined capacity of 400mAh. This setup is chosen for its simplicity ensuring manageable power for the battery management system (BMS).

3.1.6 STEP-DOWN CONVERTER

A step-down converter has the role to reduce the input voltage to a lower output voltage while maintaining the power balance. The demand for step-down DC–DC converters is increasing rapidly, looking in the field of industrial applications such as industrial robotics, the Internet of Things (IoT) and embedded systems (automotive industry) [10].

Step-down converters operate on the principle of switching regulation. The basic operation involves charging an inductor when the switch is closed, storing energy in its magnetic field. When the switch opens, the inductor releases this stored energy to the output. The average voltage output is controlled by the duty cycle of the switching signal, effectively stepping down the input voltage.

Using a buck converter is an effective solution for charging a battery with relatively

low voltage values. However, using a simple buck converter can result in power losses due to the diode and inductive filters [10]. Despite this, it remains a viable alternative for applications involving low charging currents. Here are a few examples of step-down converters, justifying the hierarchical placement of the converter I used:

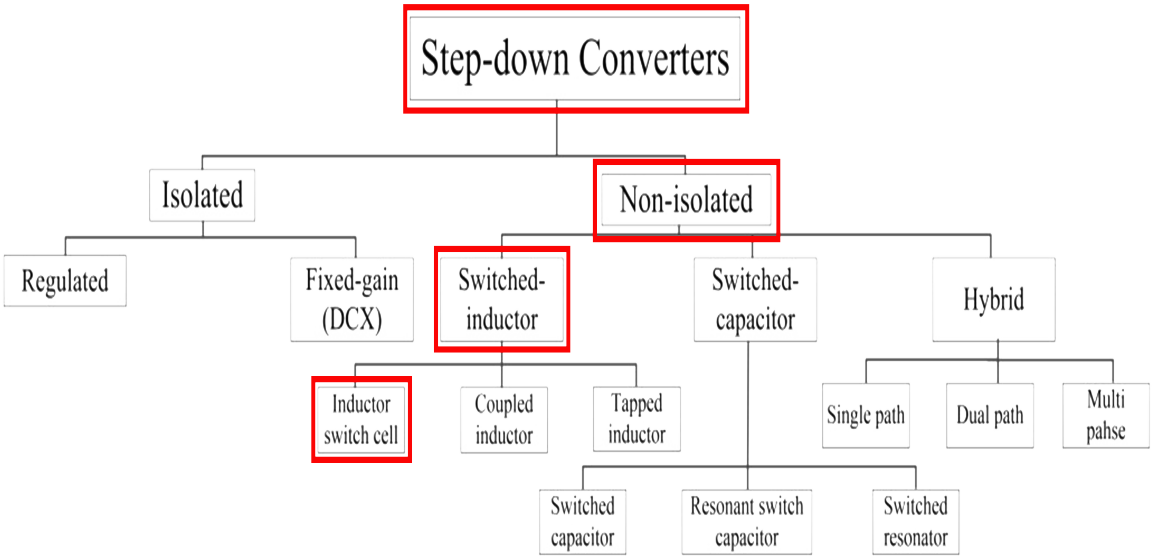

Figure 3.3: “Classification of Step-Down converters”[9]

The presence of a MOSFET and a transistor for controlling the switching action, along with an LC filter, aligns with the characteristics of an inductor switch cell converter in my project. The ESP32 controlling the PWM further confirms this classification, as it provides the necessary control signals to the switches in the converter.

3.2 SOFTWARE

3.2.1 REACT-NATIVE

React Native is an open-source framework developed for building mobile applications using JavaScript and React. It allows developers to create natively rendered mobile apps for iOS and Android, significantly reducing development time and effort.

React Native offers a wide range of libraries and plugins to meet various development needs. It can be easily integrated with services from providers like Google and Amazon to achieve versatile features, including UI elements, navigation between screens, user authentication (login/register process), sending push notifications and accessing the device’s native hardware.

React Native operates within a Node.js environment, which serves as the back-end

for executing JavaScript on the server side of mobile applications. Node.js contributes to the development of high-performance APIs that facilitate automation and data exchange.

According to Statista, „around 40% of developers have been using it during the last three years” [11]. Here is a diagram which explains the evolution of React-Native through multiple cross-platform environments:

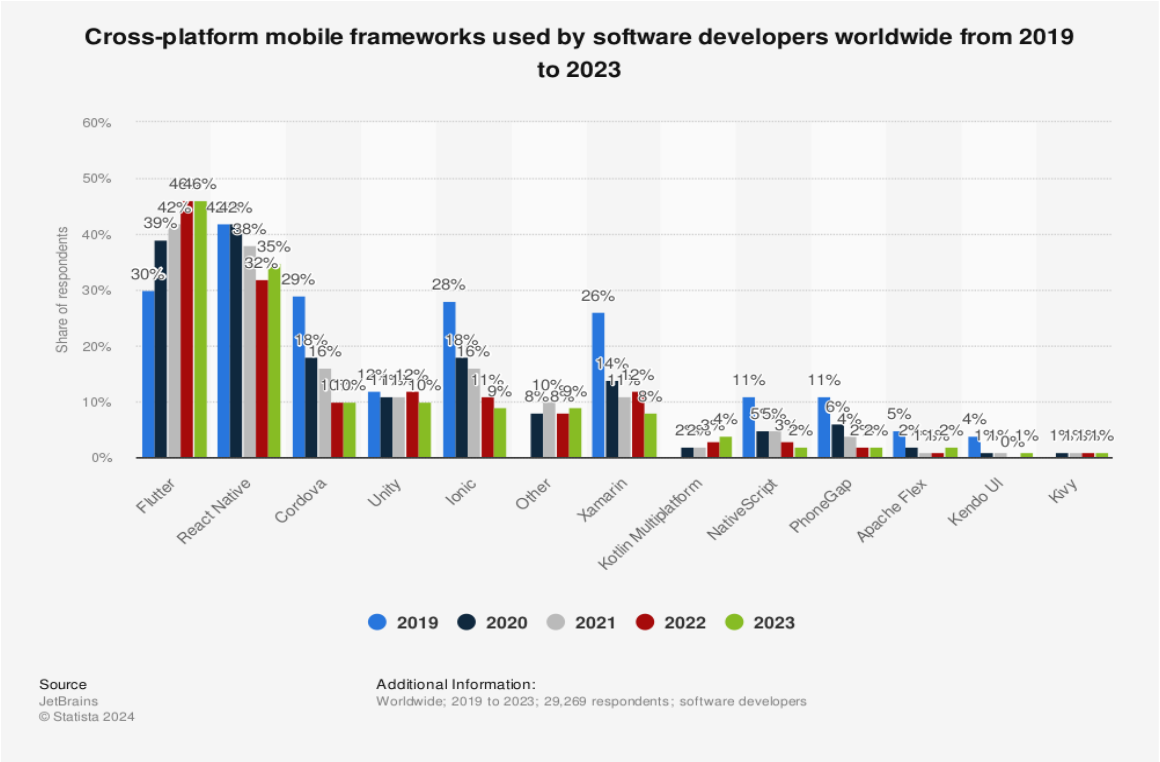

Figure 3.4: “Cross-platform mobile frameworks used by software developers” [12]

3.2.2 EXPO

Expo is a tool-chain built around React Native, designed to simplify the creation and distribution of cross-platform applications. It provides a managed build environment and tools for testing and debugging, along with a range of services for developing, building, and publishing React Native apps.

Expo includes a platform-neutral API that allows developers to access native device functions such as the camera, GPS and push notifications without writing platform-specific code. This makes it easier to create apps that function on both Android and iOS. Here are some dedicated tools that Expo offers in order to develop, test and deploy an application:

- Expo SDK: A set of libraries and services that provide access to native APIs for building apps without needing native code.

- Expo Go: A mobile app that allows developers to run and test their React Native projects instantly on their devices without needing to compile native code.
- Expo CLI: A command-line interface that simplifies project creation, development, and management, offering various commands to streamline the workflow.

Sometimes, React Native and Expo concepts are misunderstood by the developers as both technologies are strongly connected. To clarify, Expo is a framework and platform built on top of React Native that provides tools and services to simplify the development process while React-Native is a framework for building mobile applications using JavaScript and React [11]. Using each specific platform brings advantages and disadvantages. The following table will better describe these two technologies:

Table 3.1: Differences between Expo and React Native environments

Aspect	**Expo**	**React Native**
Ease of Setup	Simple setup with minimal configuration	More complex setup requiring configuration
Development Speed	Faster due to managed workflow and tools	Slower due to need for custom configurations
Testing and debugging	Built-in tools for streamlined testing	Requires separate setup for testing tools
Performance	Slightly lower due to abstraction layer	Higher performance with direct native access
Dependencies	Limited to Expo SDK version	Can use any compatible library or native module

3.2.3 FIREBASE

Firebase is a comprehensive app development platform created by Google, designed to help developers quickly build applications. It offers a variety of tools and services, including a Realtime Database, cloud storage, functions and machine learning capabilities. Firebase is particularly well-suited for mobile and web applications due to its easy integration with these platforms [13]. For my project I incorporated the following services:

Realtime Database: This NoSQL cloud database stores and synchronizes data in real-time between clients. It allows developers to create applications with immediate updates and consistent user experiences across different devices. The instant propagation of data changes makes Firebase Realtime Database ideal for applications needing live updates, such as chat applications, collaborative tools and IoT systems. In this project, the Realtime Database enables the mobile app to control the car prototype by updating the database with new commands.

Firebase Functions: This serverless framework allows developers to run backend code in response to events triggered by Firebase features. It extends application functionality without requiring server management. For example, Firebase Functions can be used to send notifications, perform complex data processing and integrate with third-party services, thereby automating workflows and enhancing app capabilities. In this project, Firebase Functions automate the system by sending notifications to the Android app whenever specific criteria are met, ensuring fast updates and responses.

3.2.4 ANDROID OPERATING SYSTEM

The Android Operating System is a widely-used mobile platform developed by Google, designed for smartphones, tablets and other devices. It is recognized for its extensive app ecosystem available through the Google Play Store. A huge advantage is that it supports a variety of hardware from different manufacturers. Android offers the possibility to create applications using Google services and is accessible for junior developers due to its simplified app creation and deployment process.

Being familiar with the React Native environment helped me develop the necessary mobile application for this operating system. Unlike other frameworks such as Android Studio, I was not constrained to work with the native development process. React Native provides the necessary features to access the device's hardware components, permissions and other functionalities that Android Studio accesses using Java, Kotlin or other dedicated programming languages.

4. DESIGN

4.1 SYSTEM ARCHITECTURE

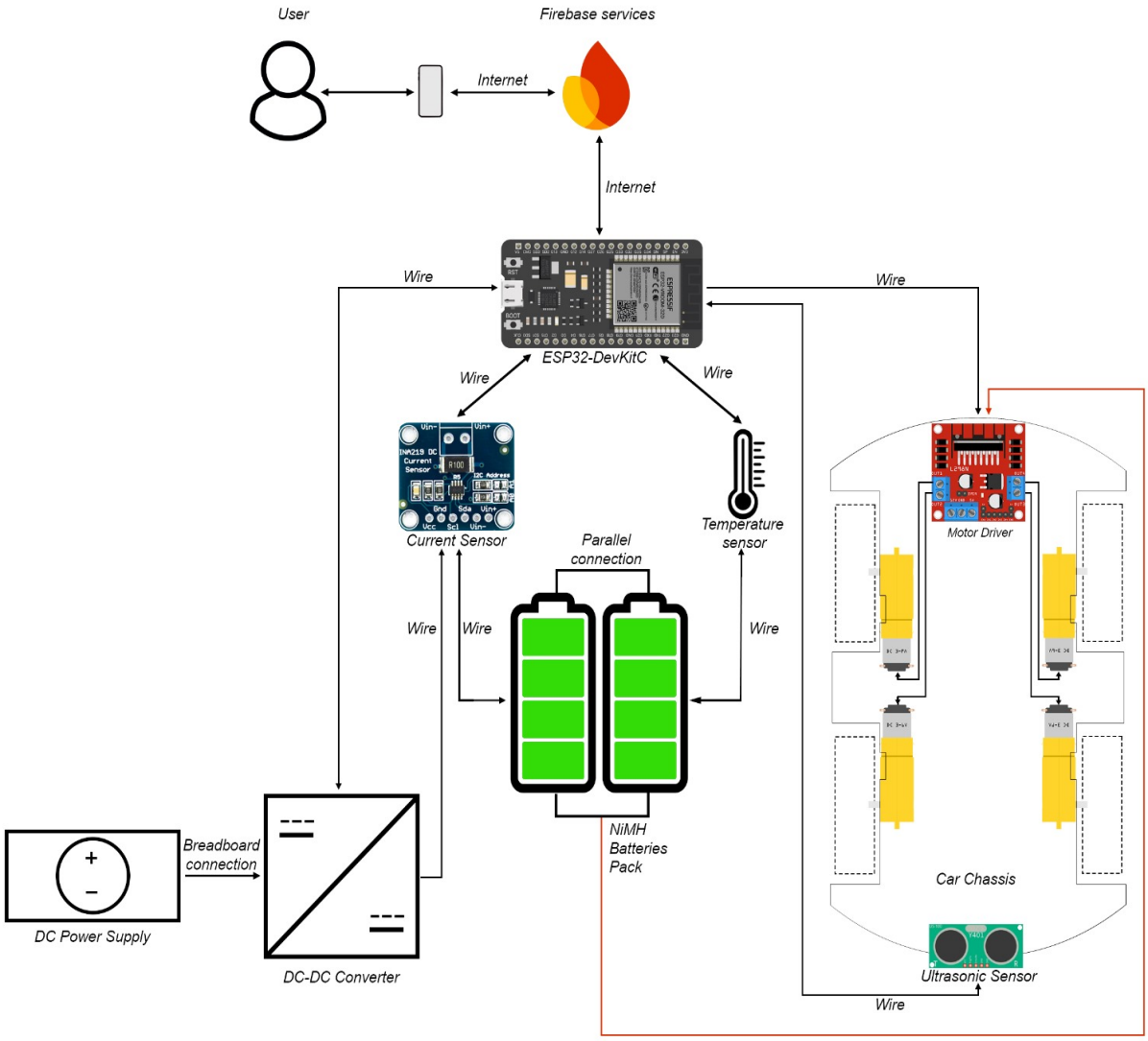

Figure 4.1: "System architecture of the Battery Management System including the vehicle control system"

In the above diagram, there is a useful schematic which explains the functioning flow of the project. It can be divided into two main subcategories:

- Battery Management System
- Vehicle Control System

4.1.1 BATTERY MANAGEMENT SYSTEM

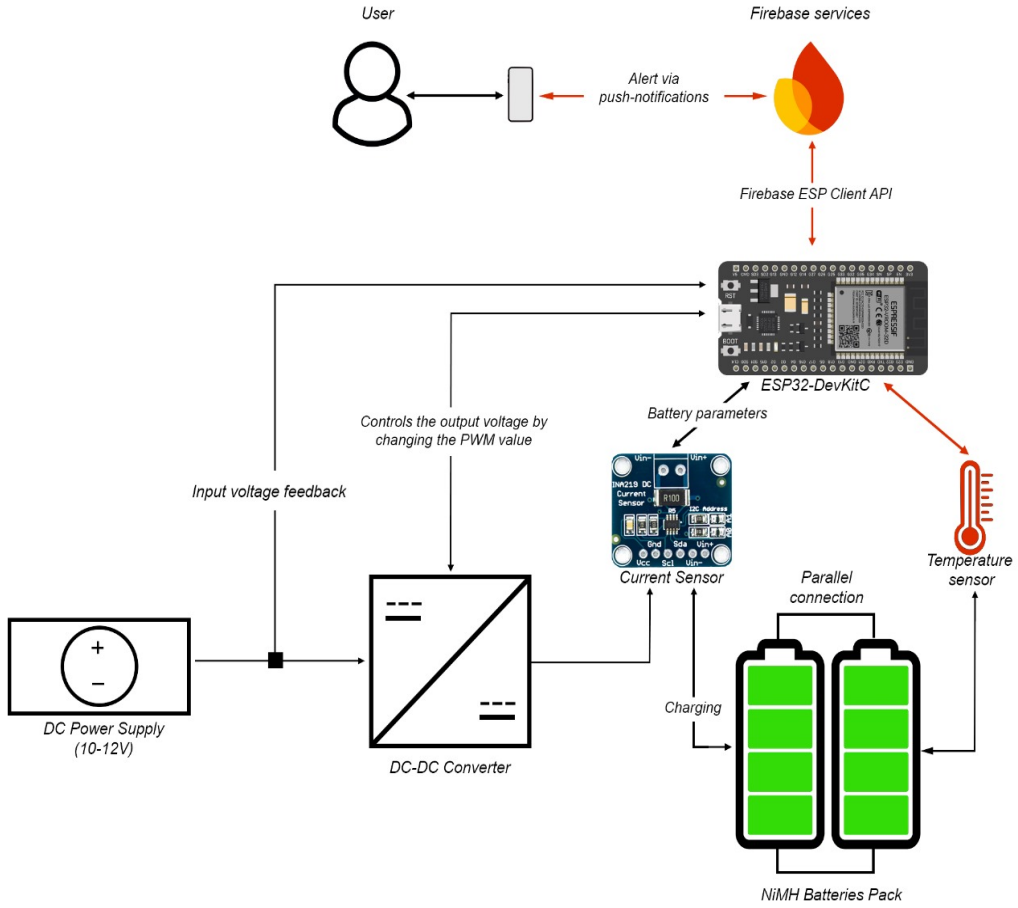

Figure 4.2: "System architecture of the battery management system"

The implemented BMS (Battery Management System) provides multiple essential functionalities:

- Continuous monitoring of the battery through an array of sensors with data accessible via a mobile application for real-time insights.
- Intelligent charging that adjusts based on the battery's current operating parameters, optimizing performance and longevity.
- User alerts through push notifications in situations where the battery is overheating, allowing for rapid intervention and prevention of potential damage.

By closely examining the system architecture diagram, we can deduce several scenarios. In the following, I will describe how each of these scenarios unfolds. The vehicle driver will have the battery status constantly monitored. If the battery percentage drops below 10%, the driver will receive a notification to alert them of the low battery level. If the battery

becomes fully discharged, the vehicle's on-board charging system offers maximum flexibility, allowing the battery to be recharged from any standard electrical outlet, notwithstanding the absence of a system to convert alternating current (AC) to direct current (DC). When the vehicle is connected to a power source, the charging process begins using a trickle-charge method, which ensures a slow and steady charge to the battery.

Throughout this process, the ESP32-DevKitC development board continuously monitors the charging parameters. This monitoring is facilitated by data sent from the INA219 current sensor and the DS18B20 temperature sensors mounted on each battery in the pack. During the charging process, there are two primary scenarios that can occur:

1. Successful charging without incidents: In this scenario, the battery is charged successfully and efficiently. The user is notified through an automated push notification once the battery has reached a full 100% charge, ensuring that they are aware their vehicle is ready for use.

2. Charging with a risk of overheating: In this scenario, if the battery temperature exceeds a predetermined threshold during the charging process, the charging will be interrupted to prevent damage or possible fire hazards. The user will receive a push notification indicating that the critical temperature level has been surpassed. Before reaching this critical point, a warning notification will be sent to alert the user of the rising temperature. The charging process will only resume if and when the temperature falls back to an acceptable level, ensuring the safety and longevity of the battery.

Additionally, all the information about battery parameters can be viewed within a mobile application, allowing the driver to closely monitor the status in real-time. There is constant feedback sent to the application's user interface regarding the power outlet being used, providing comprehensive insights into the charging process. By maintaining constant monitoring and providing timely notifications, the system ensures that the driver is always informed of the battery's status, whether it be low charge, successful charging or potential overheating risks.

4.1.2 VEHICLE CONTROL SYSTEM

The control system of the small car built in this project can be divided into two main parts:

1. The subsystem responsible for maneuvering the car.

2. The methods for driver assistance in case of danger.

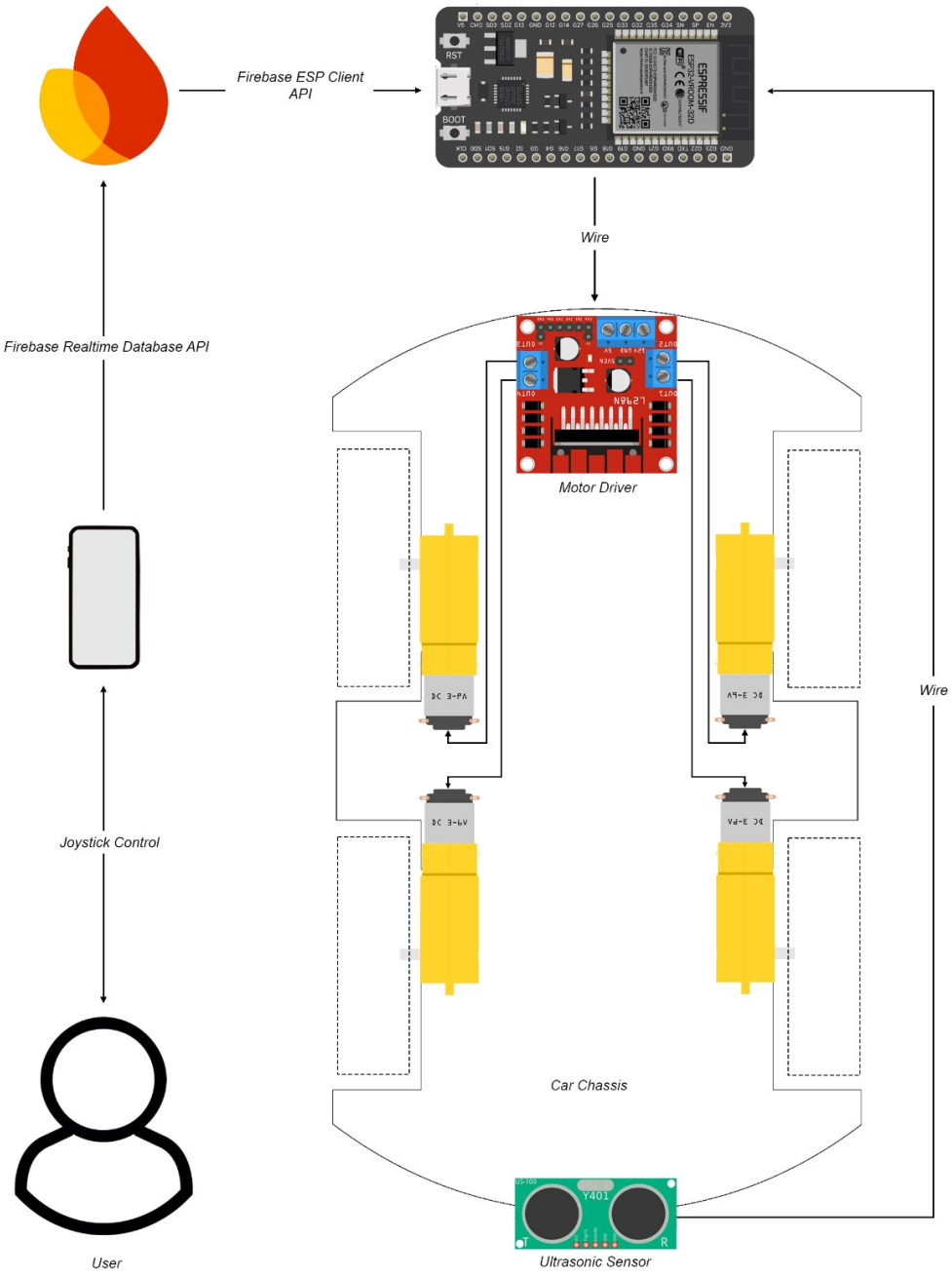

Figure 4.3: “Vehicle control system architecture [14]”

The process begins with user input from the mobile application. Within the app’s user interface, there is a dedicated section specifically for controlling the car. Based on this input, data regarding the desired direction and movement are sent to a real-time database via the Firebase Realtime Database API. From there, the information is retrieved by the ESP32-DevKitC, interpreted and transmitted to the motor driver. The motor driver is then connected with the electric motors (two motors for each motor channel of driver).

Using this information, the car can move in the following directions: forward, backward, forward-right, forward-left, backward-right and backward-left. This capability allows for a high

degree of maneuverability.

The entire experience is enhanced by the anti-collision protection provided by an ultrasonic sensor US-100 mounted at the rear of the vehicle. This sensor constantly measures the distance between the car and any objects behind. It interrupts the movement in the case an object is way too close. It is particularly useful when reversing, as it helps to avoid unwanted collisions.

4.2 HARDWARE ARCHITECTURE

In the following section, in order to present the hardware architecture, I will focus on explaining in detail the components, technologies and principles used in creating the buck converter. I will also explain how the ESP32 development board is connected and interacts with all the other hardware components within the created car.

4.2.1 STEP-DOWN CONVERTER

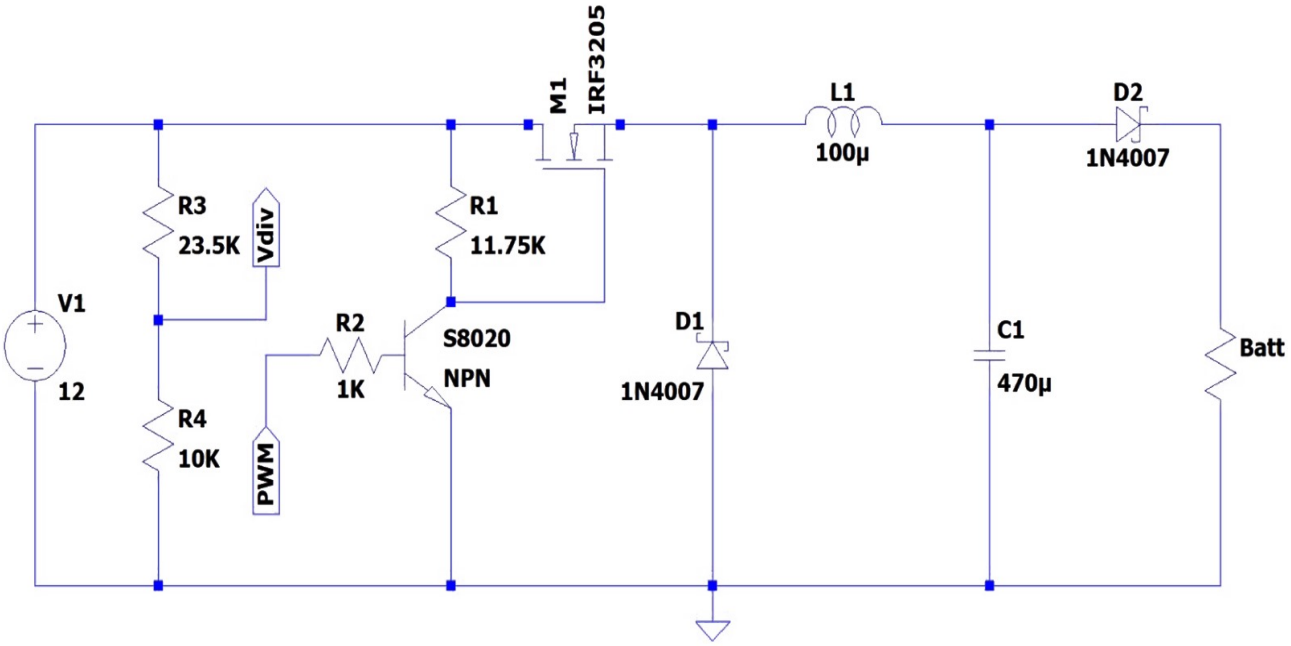

Figure 4.4: “Step-down converter architecture”

The converter is the most essential part of this project. Its purpose is to convert a higher input voltage into a lower voltage by increasing the current inside the circuit. It simulates the scenario of charging a real electric car. The design operates with an input voltage between 10 and 12V. This value meets the requirements of charging a NiMH battery, which needs a charging voltage higher than its nominal voltage (ranging between 8.4V and 9.6V for NiMH batteries).

Electrical components used:

- N-Type MOSFET “IRF3205”
- NPN Bipolar Junction Transistor “S8020”
- 2 x “1N4007” Schottky diodes
- 100 µH coil inductor
- 470 µF electrolytic capacitor
- 23.5kΩ, 11.75kΩ, 10kΩ, 1kΩ resistors

The current path within the converter is as it follows: from the power source to the MOSFET, which acts as a switch controlled by a PWM signal. By modifying the PWM value using the ESP32, a ripple is created that adjusts the voltage according to the needs of the electric car.

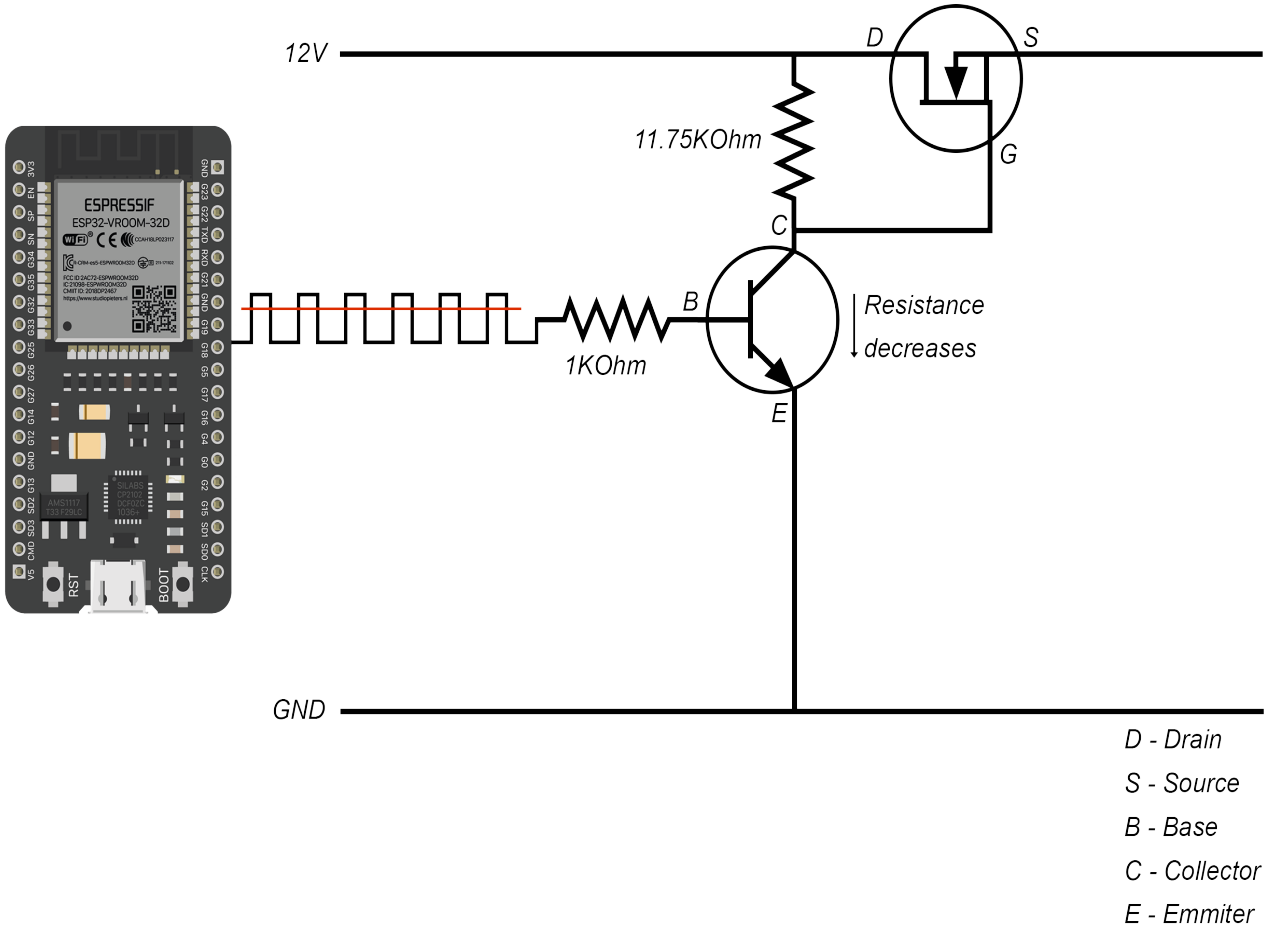

Figure 4.5: “Behavior of the system in response to high PWM value”

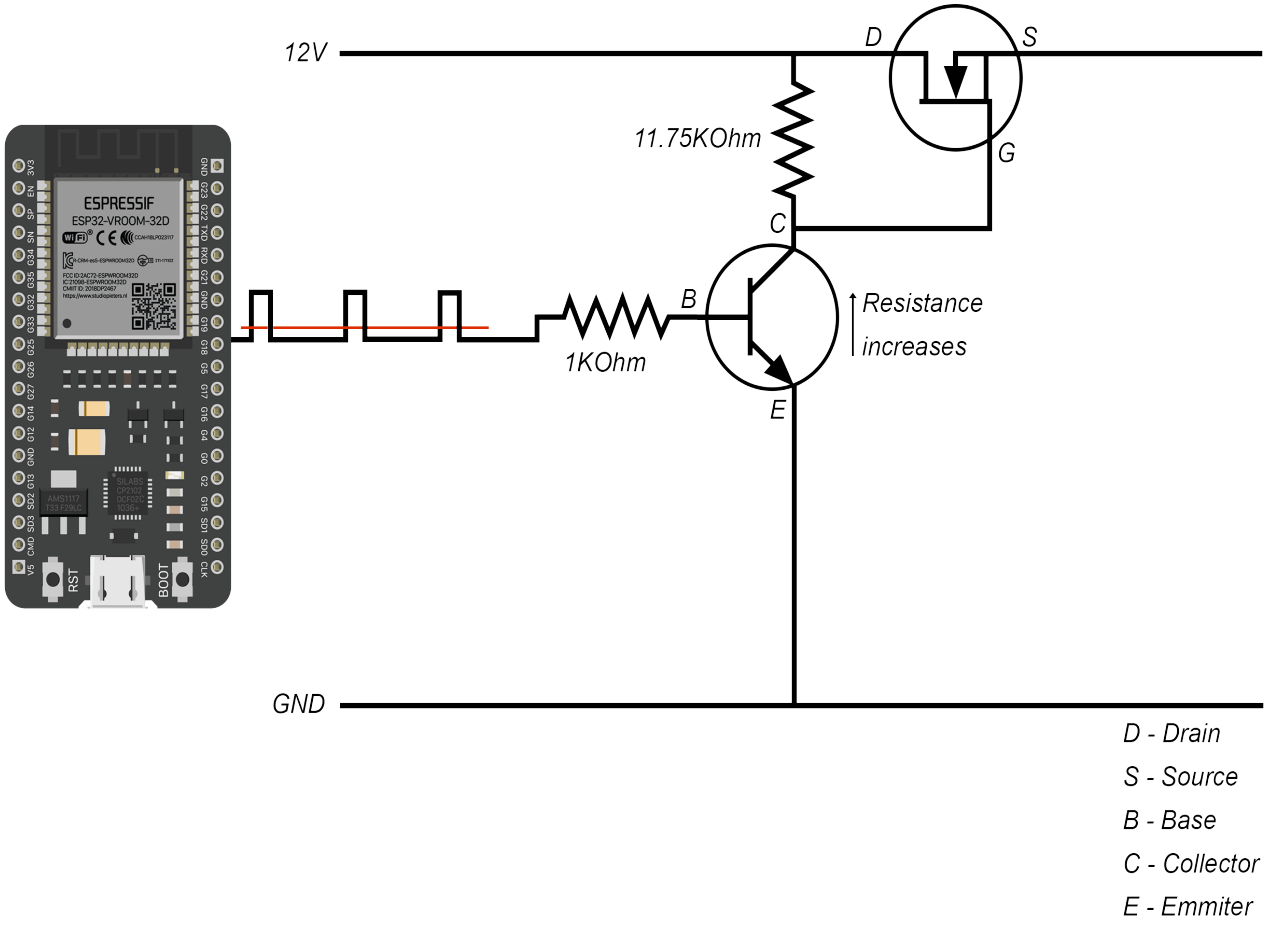

Figure 4.6: “Behavior of the system in response to low PWM value”

By using the software capabilities of the ESP32 along with its diverse pin functionality, an automated charging system can be effectively realized. In this setup, the created buck converter utilizes a PWM signal to control the charging process. The PWM signal ranges from 0 to 254, aligned with the ADC capabilities of the development board. Adjusting the PWM value allows us to alternate between the desired voltage and charging current. A lower PWM value will cause the transistor to act as a stronger resistance, resulting in a higher voltage at the gate pin of the N-Type MOSFET. This, in turn, makes the MOSFET open wider, delivering a higher output. Using a reverse analogy, a higher PWM value will have the opposite effect. Figure 4.5 and 4.6 will offer a graphical representation of both presented stages.

In the ON stage, the MOSFET switch is closed and the diode D_1 is reverse-biased (not conducting) because the voltage at the cathode is higher than at the anode. When the switch is closed, current begins to flow from the input source through the inductor (L_1) to the load. The inductor resists changes in current by generating an opposing voltage (V_L) across its terminals due to the changing current. This opposing voltage initially reduces the net voltage across the load. As the current through the inductor increases, energy is stored in its magnetic field. The voltage across the inductor can be expressed as:

$$V_L = L\frac{dI_L(t)}{dt} \tag{4.1}$$

where L is the inductance, $\frac{dI_L(t)}{dt}$ is the rate of change of current through the inductor and t is time.

In the OFF stage, the MOSFET switch is open, removing the input voltage source from the circuit. The current through the inductor starts to decrease, causing the inductor to generate a voltage that opposes the drop in current. This makes the inductor act as a current source, discharging its stored energy into the circuit.

During this stage, the diode becomes forward-biased (conducting), allowing current to flow from the inductor to the load and back through the diode. This maintains current flow through the load even though the input voltage source is disconnected. The energy stored in the inductor supports the current flow, ensuring a continuous current to the load. The voltage at the load is determined by the inductor discharging and can be expressed as:

$$V_L = -L\frac{dI_L(t)}{dt} \tag{4.2}$$

The output voltage of the MOSFET is very noisy, containing lots of switching harmonics so an LC filter is needed to stabilize it. Only after this filtering process, the converter can effectively serve its purpose and charge the batteries efficiently.

The circuit also presents a voltage divider that measures the input voltage and uses it for protection purposes, particularly to avoid fluctuations in the charging process that could damage the battery. The output voltage of this voltage divider targets to match the working voltage of the ESP32-DevKitC. The following formula is used in order to compute the accepted value:

$$Voltage_{output} = Voltage_{input}\frac{R_2}{R_1 + R_2} \tag{4.3}$$

where $Voltage_{output}$ represents the actual voltage of the power supply, $Voltage_{input}$ is the raw value read by the ADC of the board and R_1,R_2 are the resistors value. In the schematic, a second Schottky diode is included to prevent reverse charging leaks from the battery, thereby protecting the rest of the circuit.

4.2.2 ESP32-DEVKITC SUBSYSTEM

The ESP32 acts as the central processor, receiving data from various sensors:

- 2 x “DS18B20” temperature sensors for monitoring battery temperatures
- 1 x “INA219” current sensors for voltage measurement and charging current value
- 1 x “US-100” ultrasonic sensors for distance measurement to prevent accidents

- 1 x “L298N” motor drivers for controlling bidirectional motors of the chassis.
- 4 x “TT” DC gearbox motors

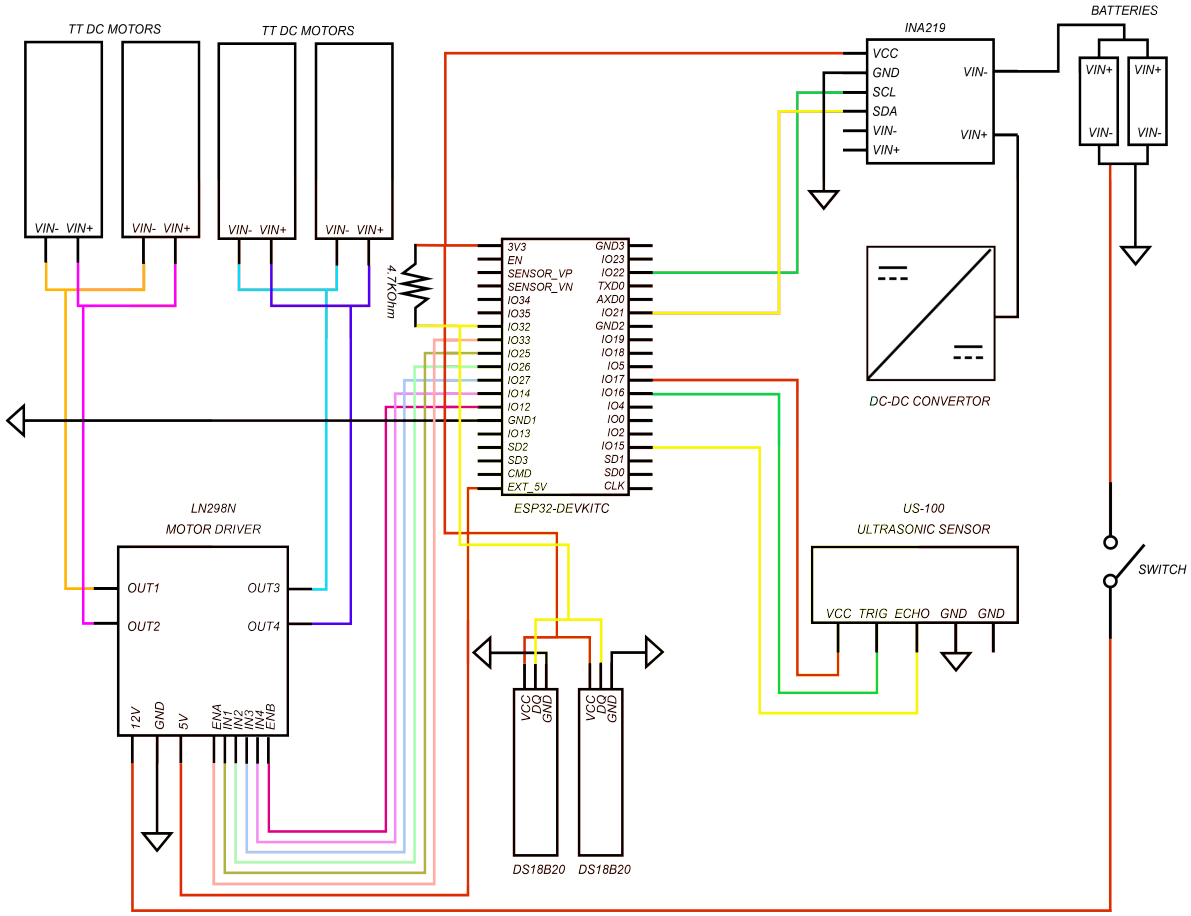

Figure 4.7: “Architecture of ESP32-DevKitC hardware subsystem”

The system architecture operates on 3.3V to minimize energy consumption. Using the ESP32 development board as the central controller for the sensor suite facilitates easier debugging and helps detect potential issues such as faulty connections, short circuits, or sensor malfunctions.

Despite these benefits, selecting the right sensors for the entire project has been a real challenge. The vast array of options available today can be overwhelming. Here are some principles used in selecting the components that form the hardware architecture: Due to the need for the entire system to operate at 3.3V, I was unable to use a common distance measurement sensor like the HC-SR04. Fortunately, within the same family of sensors, there is the US-100, which provides the necessary compatibility and operates at a voltage range between 2.4V and 5.5V. Additionally, I aimed to use a high-precision current sensor in this process. The INA219, with its integrated analog-digital converter (ADC) operating at a range

of up to 26V, was a suitable choice. The current sensor is highly efficient, measuring current through a shunt resistor with a value of approximately 0.01 Ohms

4.3 SOFTWARE ARCHITECTURE

The software architecture of this project plays an important role in ensuring the entire system functions correctly. This project is a synergy between hardware and software, meaning that the software dictates the behavior of the hardware to achieve optimal performance. This is essential for obtaining the desired behavior in both charging and sensor operation. The code can be spitted in two parts:

1. The code written for the ESP32 board.
2. The mobile application developed using an Expo project.

4.3.1 EMBEDDED CONTROL SUBSYSTEM

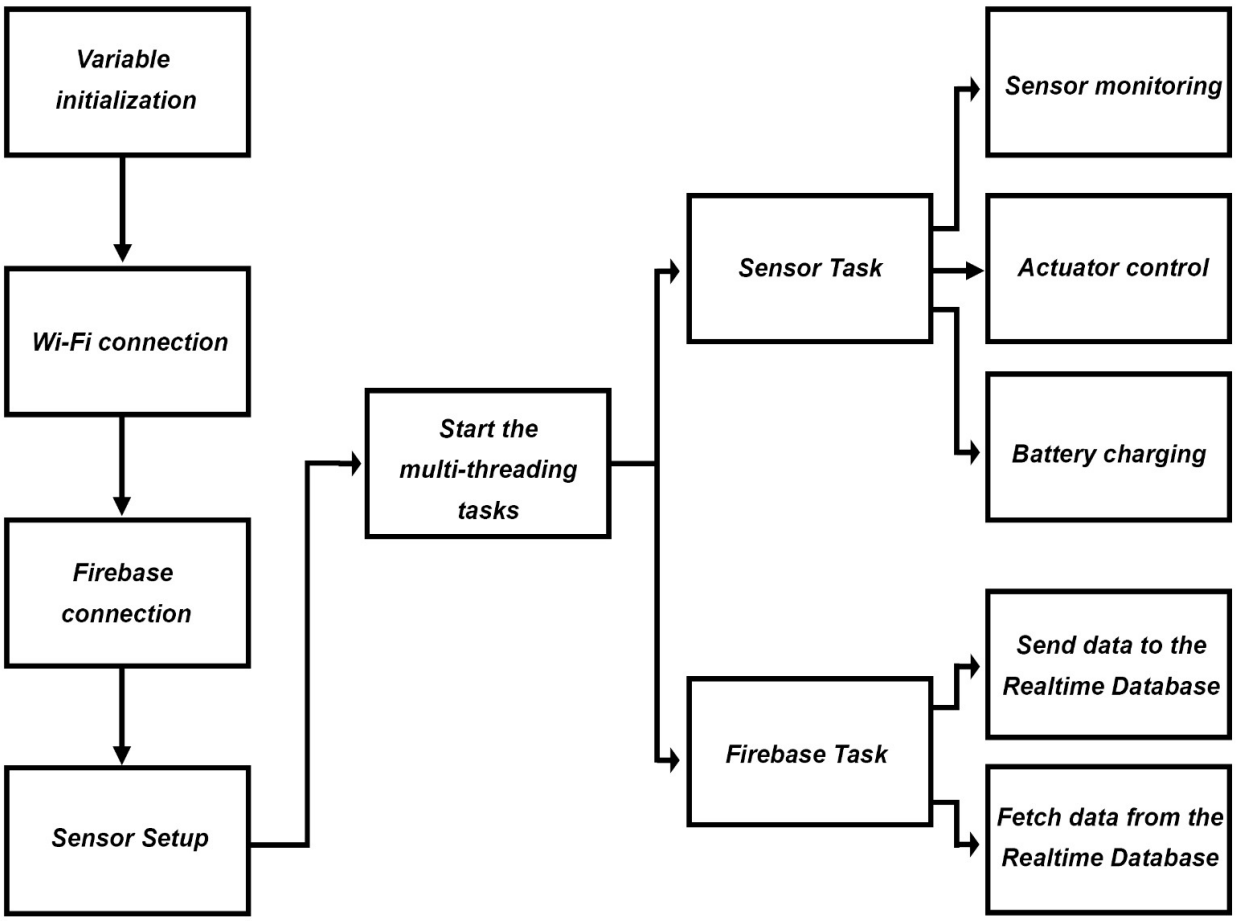

Figure 4.8: “Software processes that run on ESP32-DevKitC”

The entry step in the software field of this project is establishing an Internet connection. This connection subsequently offers access to Firebase which later provides multiple features to the system. Without this initial connection, none of the subsequent processes would be possible, making it the foundational step in the entire setup.

Initializing the real-time database is crucial within this architecture. The Firebase real-time database acts as a "buffer zone" between the board and the mobile application, managing the high data flow that the ESP32 sends or receives. Without successfully establishing this connection, the entire system's purpose would be compromised. To achieve this connection, the Firebase ESP Client library will be used, which handles the necessary configurations for connecting to Firebase. Authentication is done using a token, saved in the API configurations once the database URL and API key are validated. Upon successful authentication, the connection is established with the complete credential configuration.

To fully utilize the potential of the development board, a multi-threaded approach is adopted. During the early stages of the project, it was observed that executing and implementing the entire code using a single thread was highly inefficient. This inefficiency not only reduced the effectiveness of the buck converter but also impacted the car's movement system. Therefore, a multi-threaded approach was necessary to enhance the system's performance and responsiveness.

Currently, the software architecture is managed using two threads, which is the maximum number of threads supported by the ESP32. sensorTask and firebaseTask are the responsible threads in which the essential code of the control unit is encapsulated in.

The sensorTask is responsible for collecting and monitoring sensor values and making relevant decisions based on them. On the other hand, the firebaseTask focuses on sending and receiving new data to and from Firebase RTDB. This dual-thread approach ensures that both data collection and communication with the database are handled efficiently and effectively, thereby optimizing the overall system performance.

Incorporating a buck converter into a Battery Management System (BMS) like mine offers several advantages, especially for efficient battery charging. To initiate the charging process, the ESP32-DevKitC must first verify if the car is connected to a valid power supply. Under normal conditions, when the battery is not charging, the voltage on the feedback bus of the power supply should be close to or equal to 0. Once a voltage greater than 3.3V but less than 12V is detected, the charging process can begin. This precaution helps protect the battery from harmful voltage fluctuations that can reduce its lifespan.

Heat is a major concern when charging batteries, as it can significantly affect performance. Research indicates that it is not the charging power itself but the heat generated from using a specific charging power that impacts battery performance. This is particularly true for batteries charged with higher currents and voltages.

For the NiMH batteries used in my project, temperature sensors continuously monitor the battery temperature. The measured data is sent to the board for processing. The charging process will not begin if any of the batteries do not meet the required temperature conditions.

An underlying assumption of the system is that while the battery is charging, the car should remain stationary. This mirrors real-world scenarios where a driver would not move the vehicle while it is connected to a charging station or a normal AC outlet. Consequently, task priorities are set based on this assumption.

The following diagram illustrates how the software manages these various use cases related to the car prototype:

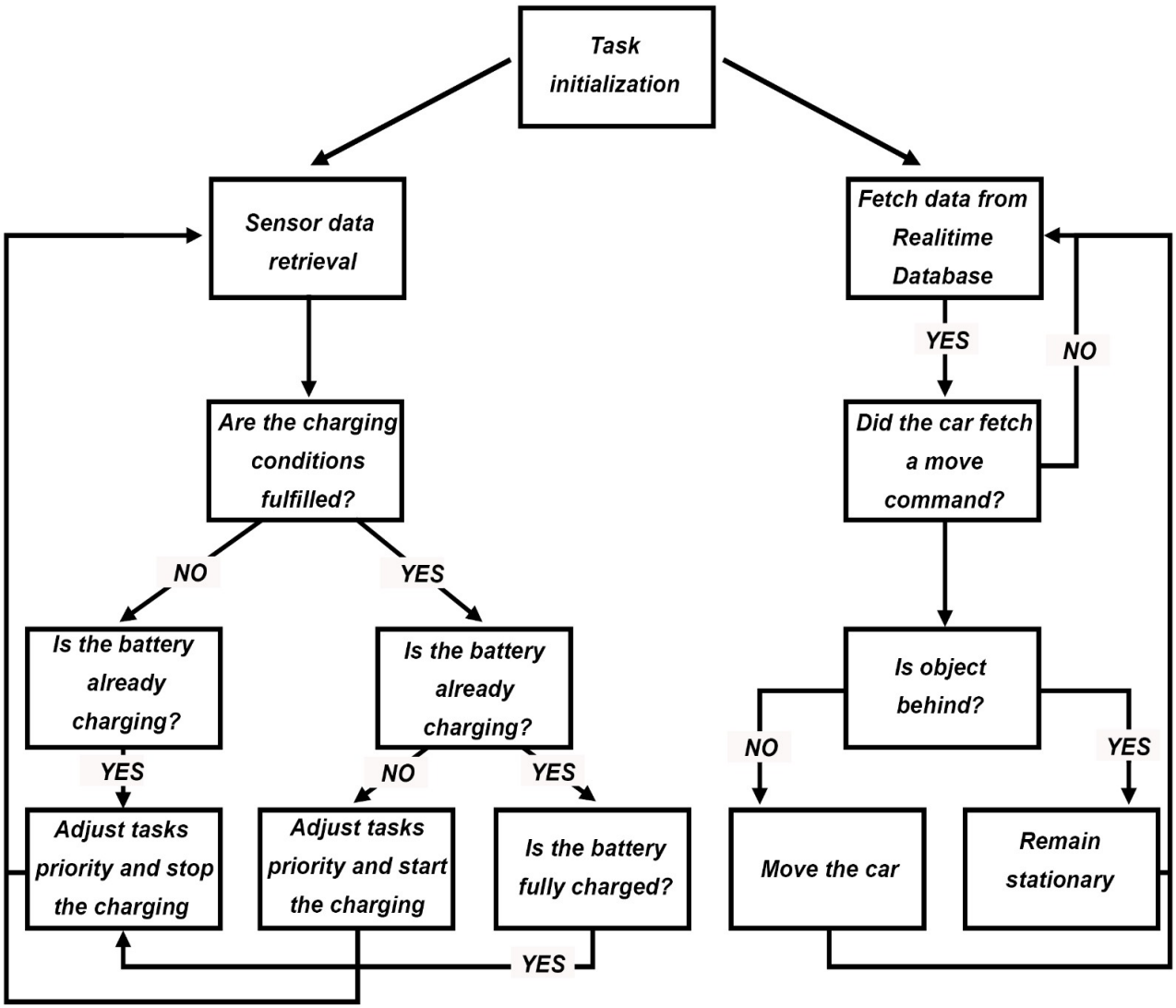

Figure 4.9: "Use cases managed by the software architecture"

4.3.2 MOBILE APPLICATION

In this chapter, I will discuss about the functionality of the mobile application developed for monitoring and controlling the complete set of components in the electric vehicle, as well as the battery management system.

The mobile application is designed to handle the actuator control at the electric vehicle level and battery parameter monitoring. To achieve the desired functionality, the application will utilize technologies provided by Expo and React Native in combination with the Firebase API.

For this project, I opted to have Expo manage all the application's resources. This decision allows the Expo tools to handle the heavy lifting, requiring me only to write JavaScript code. The Expo Go app utilizes the Expo Runtime to offer a real-time preview of the application, so I don't need to concern myself with native code or native modules.

To clearly explain the overall software architecture of my application, I will use a well-established architectural design pattern: MVC (Model-View-Controller).

I will begin with the View component, which typically refers to the graphical representation and the way elements are presented to the user. In my mobile application, the Views encompass all graphical elements on each screen. This includes images, texts and shapes that contribute to the application's aesthetic appeal.

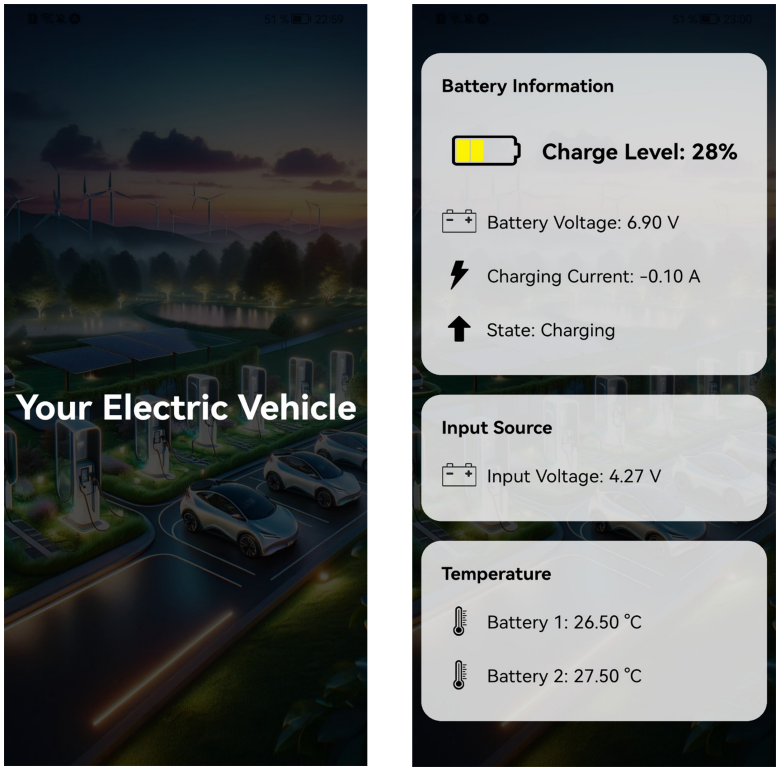

Figure 4.10: "View components found inside EV's mobile application"

In addition to the presence of View components, which strongly influences the design of the mobile application, we also encounter control elements or so-called Controllers. These act as intermediaries between the Views and the Model. Often, these components can modify, delete or add data at the Model level. They are most commonly found in the form of buttons, text boxes or input elements. In my application, there are only two such components:

- Buttons: facilitates communication between the screens of the application.
- Joystick: enables and controls how the electric car move.

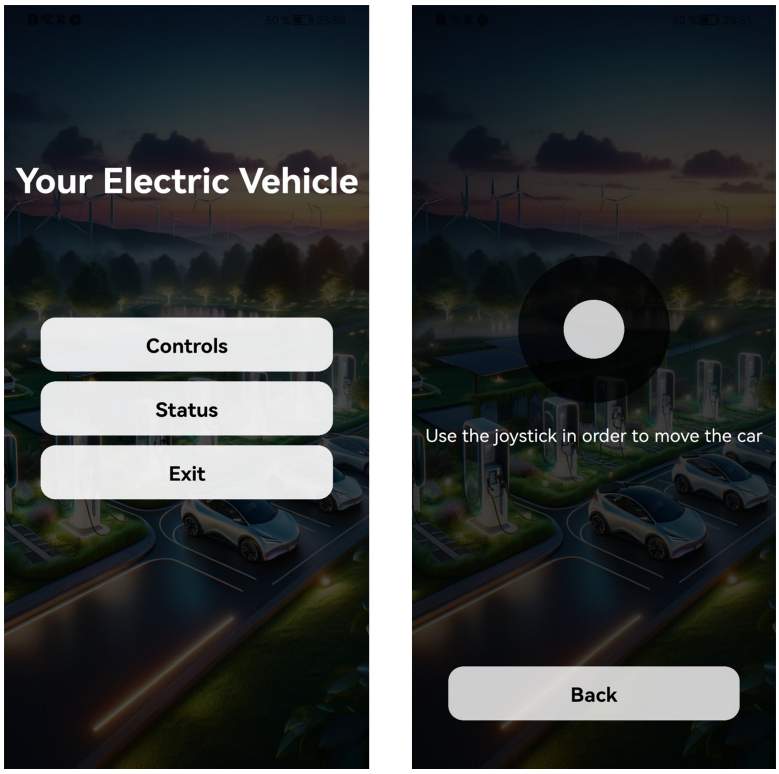

Figure 4.11: "Controllers found insider EV's mobile application"

The Model is the component responsible for managing the data within the application. Most often, this component interacts with the application's database, playing the role of storing, updating and synchronizing data. The Model component in this mobile application play the following roles:

1. Motor Control Data : manages the control signals sent to the motors of the small electric vehicle.
2. Battery Parameters : maintains up-to-date information on the battery parameters received from the Battery Management System (BMS).
3. Firebase Services : handles the communication with Firebase, which serves as the intermediary between the mobile application and the ESP32 development board. Model component ensures that data is correctly sent and received

4. Notifications: The Model manages notifications sent via Firebase Functions, alerting users about relevant events

As the image 4.12 shows, Expo uses a standard project configuration when initializing a new project. Below, I will briefly describe the role of the components in my project and what their purpose is.

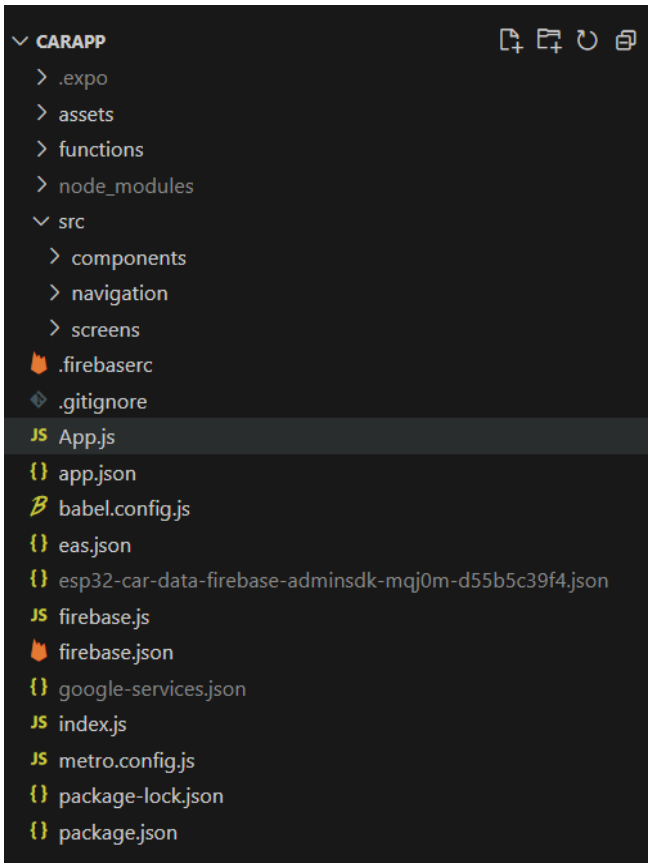

Figure 4.12: “Mobile application project structure”

- .expo/: Contains configuration files and metadata of the Expo environment.
- assets/: Stores images and or other graphical resources used in the project.
- functions/: Contains serverless functions for Firebase backend services in order to provide push-notifications for the application
- node-modules/: Contains all the npm packages installed for the project.
- src/:
  - components/: Dedicated for reusable UI components.
  - navigation/: Contains navigation-related code such as navigators and route configurations.
  - screens/: Stores the different screen components for the application, each representing a distinct UI view or page.
- App.js: The main entry point of the application where the root component is defined.

- app.json: Configuration file for the Expo project, including app name, slug and other settings.
- babel.config.js: Babel configuration file, used for setting up JavaScript transpiling.
- eas.json: Configuration for Expo Application Services (EAS) which manages build and deployment processes.
- esp32-car-data-firebase-adminsdk.json: Firebase Admin SDK credentials for connecting to Firebase services securely.
- firebase.js: Contains Firebase initialization code and configuration for connecting the app to Firebase services.
- firebase.json: Configuration file for Firebase hosting and deployment settings.
- google-services.json: Contains configuration information for integrating Firebase services in Android apps.
- index.js: The entry point for registering the root component with the React Native app.
- metro.config.js: Configuration file for Metro, the JavaScript bundler used by React Native.
- package.json: Contains data about the project, including dependencies, scripts.

Most of the code is centered around the “App.js” file, Firebase-specific files and directories and the files within the “src/” directory. Here’s a detailed description of how the entire system operates:

The mobile application consists of three main screens: MainScreen, StatusScreen and ControlScreen. Each screen has a specific function:

1. MainScreen: This is the initial UI displayed when the application starts. It serves as an intermediary between the other two screens. It includes three buttons: "Controls”, "Status" and "Exit". These buttons navigate to the remaining screens, StatusScrenn, ControlScreen respectively allow exit from the application when pressed..
2. StatusScreen: This screen displays real-time data information from the Battery Management System (BMS). It is divided into several sections, each one focused on different categories of parameters. It has a back button which offers the possibility to return to the main screen.
3. ControlScreen: This is the control interface for the electric car. It includes a joystick controller and a button to return to the main interface as well.

As I previously mentioned, this mobile application uses Firebase Google services for the Realtime Database connection and Firebase Functions which enable a fully automated system that sends notifications based on specific events.

Regarding the connection with Firebase, it can be viewed from two perspectives: the server-side connection and the client-side connection.

Firebase server-side connection is established to utilize Firebase's serverless services. This means that the developer can write backend code directly without needing to set up their own server. This can be very advantageous, as it eliminates concerns about scaling resources and users demand. A client-side connection involves establishing a link to Firebase, managing push notifications and saving data to Firebase Realtime Database (RTDB). The code uses Firebase's Realtime Database API to read and write data.

This project utilizes two key notification protocols: the Expo Notifications API for handling push notifications and sending POST requests to the HTTP/2 API. Both protocols offer advantages, such as providing all the necessary client-side functionality for push notifications and not requiring any authentication method [15].

The process of transitioning from one screen to another is called navigation and it primarily uses the React Navigation library. This library offers several methods to achieve navigation but in this project, the main focus is on Stack navigation. The operation principle is straightforward, utilizing two methods in the background: push and pop which add or remove screens from the navigation stack. Interaction with all the graphical elements was possible using this feature. Figure 16 is used to explain the entire process:

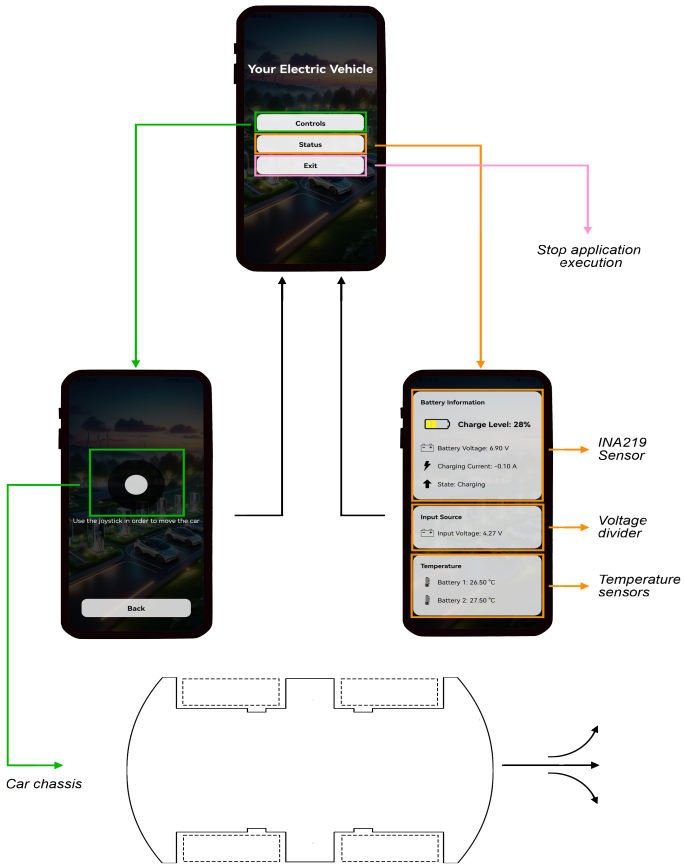

Figure 4.13: "Mobile application workflow"

5. IMPLEMENTATION

Given that the implementation section will primarily focus on how the code contributed to the realization of the entire system, it will be divided similarly with the software architecture: Embedded Control Subsystem and Mobile Application.

5.1 EMBEDDED CONTROL SUBSYSTEM

The embedded control part of the project begins with connecting to the internet, followed by establishing a connection to the Firebase Realtime Database. This order is essential because maintaining a constant internet connection is required for accessing Firebase services.

```
void connectToWiFi() {
  WiFi.begin(WIFI_SSID, WIFI_PASSWORD);
  Serial.print("Connecting to WiFi");
  while (WiFi.status() != WL_CONNECTED) {
    Serial.println("Connecting...");
    delay(100);
  }
  Serial.println("\nConnected to WiFi");
}
```

Figure 5.1: “ESP32-DevKitC internet connection”

`begin()` method utilizes the user credentials definition (`WIFI_SSID` and `WIFI_PASSWORD`) to initiate an Ethernet connection with the specified network.

```
void connectToFirebase() {
  config.api_key = FIREBASE_AUTH;
  config.database_url = FIREBASE_HOST;
  if (Firebase.signUp(&config, &auth, "", "")) {
    Serial.println("Connection established!");
    signupOK = true;
  } else {
    Serial.printf("%s\n", config.signer.signupError.message.c_str());
  }

  config.token_status_callback = tokenStatusCallback; // see addons/TokenHelper.h

  Firebase.begin(&config, &auth);
  Firebase.reconnectWiFi(true);
}
```

Figure 5.2: “Firebase RTDB connection”

To connect to the Firebase, the ESP32 needs the API key and the Database URL,

which are prerequisites for initiating the authentication process. This data can be obtained using FirebaseConsole.

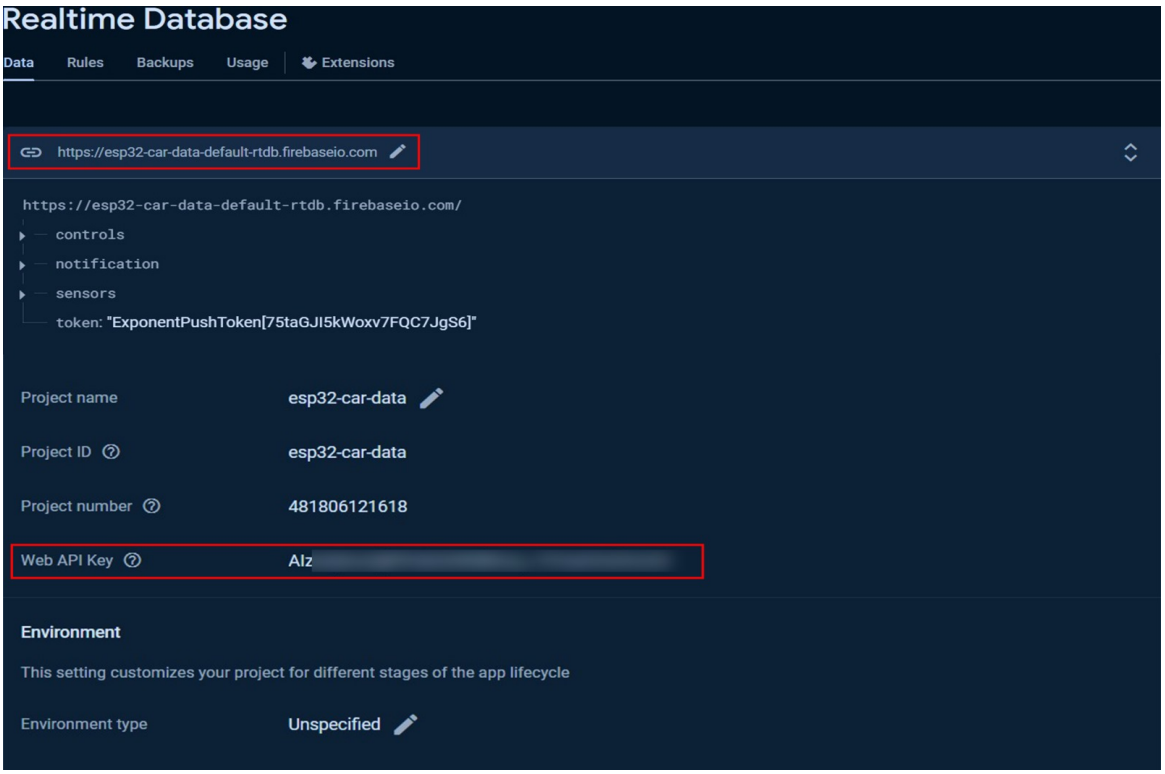

Figure 5.3: "Firebase console providing necessary credentials"

If the provided credentials are correct, a token is generated and saved in the ESP32's configuration files. Once the token is acquired, the connection to Firebase is established and maintained.

The following step involves configuring the pins to establish connections with the necessary sensors. In this section, I won't go into detail about each individual connection but will emphasize the most important details. To measure the input voltage of the realized buck converter, a voltage divider is used. This voltage divider sends a signal to the ESP32 development board. The ADC (Analog-to-Digital Converter) on the board must be configured to accurately convert this signal based on its characteristics. The voltage divider maps the input voltage to a range between 0 and 3.3V. To interpret this signal and convert it to a digital value, the following configurations are applied:

```
adc1_config_width(ADC_WIDTH_BIT_12);  // Set ADC resolution to 12 bits
adc1_config_channel_atten(ADC1_CHANNEL_0, ADC_ATTEN_DB_11);  //0-3.6V range
```

Figure 5.4: "ESP32 ADC configuration"

`adc1_config_width(ADC_WIDTH_BIT_12)` sets the ADC resolution to 12 bits,

meaning a higher precision when converting the analog values to digital (2^{12} = 4096 possible values).

`adc1_config_channel_atten(ADC1_CHANNEL_0, ADC_ATTEN_DB_11)` specifies that the configuration applies to channel 0 of ADC1 which is mapped to GPIO36 on the ESP32. It configures the attenuation used to allow the ADC to read higher input voltages. With an 11dB attenuation, the ADC input range is extended to 0-3.6V instead of the default 0-1.1V. This means the ADC can handle higher input voltages without being damaged and it will correctly map these higher voltages to the 12-bit digital value range.

To regulate the output voltage and current of the buck converter, a PWM signal is employed. The function `ledcAttachChannel(PWM, frequency, resolution, pwm_channel)` is used to establish the signal's resolution and frequency and to assign the GPIO pin for its output. This function call also requires specifying a channel configuration, with `pwm_channel` determining the specific channel used in this context.

Effectively utilizing and managing the resources of the development board is one of the main focuses when writing embedded code. Multi-threading techniques played a significant role in handling all the software functions within the project's system. As outlined in the software architecture chapter, `sensorTask` and `firebaseTask` were established to enhance code organization and resource management. The following snippet illustrates their configuration:

```
xTaskCreatePinnedToCore(
  sensorTask,               // Task function
  "Retrieve sensors data", // Name of the task
  4096,                     // Stack size
  NULL,                     // Task parameter
  2,                        // Priority
  &sensorTaskHandle,        // Task handle
  0                         // Core where the task should run
);

xTaskCreatePinnedToCore(
  firebaseTask,       // Task function
  "Firebase Task",    // Name of the task
  8192,              // Stack size
  NULL,               // Task parameter
  2,                  // Priority
  &firebaseTaskHandle,// Task handle
  1                   // Core where the task should run
);
```

Figure 5.5: "Task initialization"

`xTaskCreatePinnedToCore()` is a function provided by the FreeRTOS operating system which helps create a new thread of execution. It has the possibility to attach it to a specific core of the processor, manage the allocated memory, and even set the task priority.

The `sensorTask` has an allocated stack size of 4096 words (with each word being 2 bytes) and is assigned a priority level of 2. Meanwhile, the `firebaseTask` is given a stack size of 8192 words, with the same level of priority. Notably, each task runs on a separate core of the ESP32-DevKitC processor, utilizing both available cores.

From now on, the focus will be on describing the tasks presented earlier. We will start with the task responsible for the communication between Firebase RTDB and the ESP32.

```
void firebaseTask(void *parameters){
  while(1){
    if (Firebase.ready() && signupOK) {
      // Sending data to Firebase
      if (millis() - lastSendTime > 5000 || lastSendTime == 0) {
        lastSendTime = millis();
        Firebase.RTDB.setFloat(&data, "/sensors/input_voltage", input_voltage);
        Firebase.RTDB.setFloat(&data, "/sensors/iBatt", iBatt);
        Firebase.RTDB.setFloat(&data, "/sensors/vBatt", vBatt);
        Firebase.RTDB.setFloat(&data, "/sensors/tBatt1", temperature_batt1);
        Firebase.RTDB.setFloat(&data, "/sensors/tBatt2", temperature_batt2);
        Firebase.RTDB.setInt(&data, "/sensors/charging", isCharging);
        Firebase.RTDB.setInt(&data, "/sensors/power", power);
      }

      // Receiving data from Firebase
      for (int i = 0; i < 4; i++) {
        if (Firebase.RTDB.getInt(&data, control_keys[i])) {
          if (data.dataType() == "int") {
            control_values[i] = data.intData();
          }
        }
      }
      handleControls(control_values);
    } else {
      Serial.println("Firebase not ready!");
    }
    vTaskDelay(pdMS_TO_TICKS(100));
  }
}
```

Figure 5.6: “Dedicated Firebase task”

In this task, the database status will be checked for each iteration of the code by using the `ready()` function and verifying if the connection to the database was successful. `ready()` function needs to be called to handle authentication tasks, its main purpose being token status generation.

The workflow for this task is as follows: periodically, at a predetermined interval, sensor data is sent to the RTDB. This allows the information to be forwarded to the mobile application. Sending data at fixed intervals is justified by the need to optimize the algorithm, thereby avoiding unnecessary consumption of the processing power provided by the board’s processor while the safety requirements are still met. Concurrently, without being limited by the number of code executions, data is fetched from the RTDB. Based on this data retrieval,

an interpretation is made regarding the direction of the car's movement.

As shown in figure 5.6, a `control_values` array is created to locally store the movement command from Firebase. The Firebase database contains multiple nodes but the node of interest here is the `controls` node, which contains the movement commands. These commands are encoded as follows: `control_down`, `control_left`, `control_right`, and `control_up`. By iterating through all the fetched values, a specific movement command is generated. `handleControls()` function processes all the control field data as it is shown in the diagram:

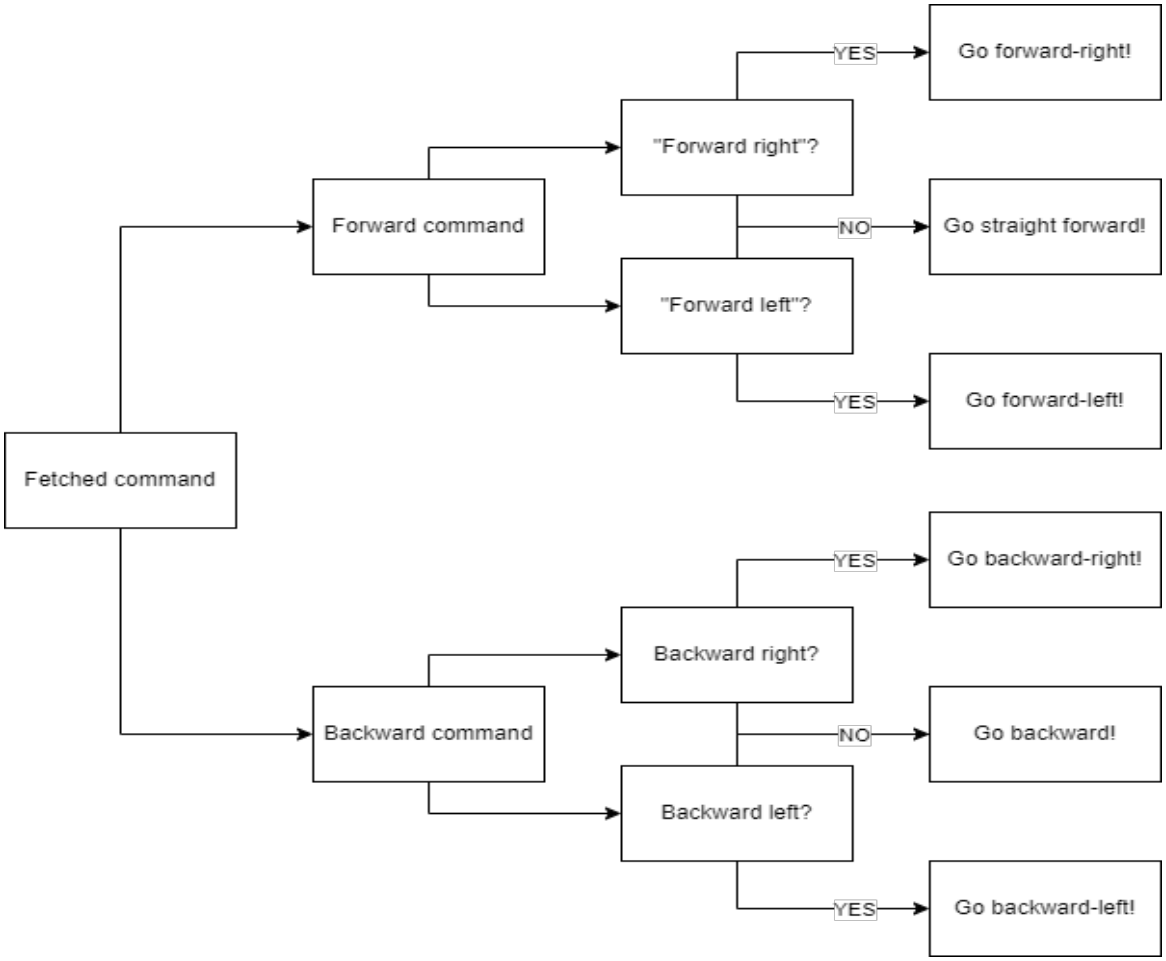

Figure 5.7: "`handleControls()` function workflow"

Within this function, the code for collision avoidance when reversing is implemented. When the embedded system logic detects a command from the RTDB which says to go backward (whether it's a command for backward-right, backward-left or straight backward), the ultrasonic sensor activates, preventing the vehicle from moving if the distance to a potential object is less than 8 cm.

The implementation for sensorTask fulfills the following functions: it reads data from the sensors and controls the battery charging process using the charging algorithm. The most important data points from the implemented Battery Management System are the

charging current, the battery voltage and the input voltage from the power source. Using these parameters, the charging system can adapt to the battery's requirements. The process of reading these parameters is as follows.

```
input_voltage = getInputVoltage(10, 3.3);
iBatt = ina_219.getCurrent_mA();
vBatt = ina_219.getBusVoltage_V() + (ina_219.getShuntVoltage_mV() / 1000);
power = ina_219.getPower_mW();
```

Figure 5.8: "Battery parameters reading and computation"

The `getInputVoltage()` function in the provided code reads the input voltage from an analog sensor using the ESP32's `adc1_get_raw()`. It takes a specified number of samples to average the readings for accuracy. Within a loop, it retrieves raw ADC values from channel 0, converts them to voltage using the reference voltage of the ESP32 architecture and the divider ratio (based on the voltage divider formula), and accumulates these values. After averaging them over the specified number of samples, it returns the calculated average voltage.

```
float getInputVoltage(int n_samples, float ref_v) {
  float voltage = 0.0;

  for (int i = 0; i < n_samples; i++) {
    int raw_adc = adc1_get_raw(ADC1_CHANNEL_0);  // Read the raw ADC value from channel 0, GPIO36
    voltage += (raw_adc * ref_v / 4095.0) / divider_ratio;
  }
  voltage = voltage / n_samples;
  return voltage;
}
```

Figure 5.9: "`getInputVoltage()` function"

`getBusVoltage_V()` returns the voltage at the V+ terminal of the INA219 module, representing the bus voltage in volts. `getShuntVoltage_mV()` returns the voltage across the in-built shunt resistor in millivolts (mV). This value is divided by 1000 to convert it to volts before adding it to the bus voltage. Together, they provide the total voltage across the battery (`vBatt`). `getCurrent_mA()` reads the current flowing through the circuit connected to the INA219 module. The result is assigned to the `iBatt` variable.

Obtaining the temperatures of each battery using the `DallasTemperature` library and its methods for extracting temperature values proved to be intensive in terms of processing resources. To maintain the algorithm's effective utility, a technique was needed to reduce the frequency of its execution. The proposed solution was to use a timer to request temperatures at longer intervals, ensuring that the `requestTemperatures` function is not called constantly but only once every two seconds. The `requestTemperatures` function takes a long time

to execute because it initiates a temperature conversion on the Dallas sensor, which involves reading the temperature from the physical sensor. This operation is relatively slow because the Dallas sensor uses a 1-Wire serial communication protocol, which is slower than other communication protocols. Additionally, Dallas sensors require a conversion time that varies based on the set resolution; higher resolutions result in longer conversion times.

The following algorithm is designed to charge the battery based on extracted parameters:

```
//value of voltage needed to start the charge
if (input_voltage > 3.3 && input_voltage < 12){
  //temperature of the baterries "ok"
  if(temperature_batt1 < max_temperature && temperature_batt2 < max_temperature){
    isCharging = true;
    adjustTaskPriority(isCharging);
    // Adjust task priorities based on charging status
  }
} else {
  isCharging = false;
  adjustTaskPriority(isCharging);
  // Adjust task priorities based on charging status
}

// PWM control logic
if(isCharging){
  if (expected_charge_current > iBatt) {
    pwm = constrain(pwm+1, 0, 254);
  }

  if (expected_charge_current < iBatt) {
    pwm = constrain(pwm-1, 0, 254);
  }

  if (temperature_batt1 > max_temperature || temperature_batt2 > max_temperature || vBatt >= max_voltage) {
    Serial.println("Battery temperature too high or battery fully charged! Stopping charging.");
    pwm = 0;
    isCharging = false;
  }

  ledcWrite(PWM, pwm);
} else {
  pwm = 0;
  ledcWrite(PWM, pwm);
}
printData();
```

Figure 5.10: “Charging algorithm implementation”

As prerequisites, the charging process is only initiated if the input voltage value of the power supply is between 3.3V and 12V. Also, both battery temperatures must remain below the maximum threshold of 50°C.

If these conditions are met, charging is enabled by setting `isCharging` variable to true and task priorities are adjusted to give higher priority to the `sensorTask` during charging. If the charging current (`iBatt`) is less than the expected charge current of 20mA, the PWM value is incremented to increase the charging current. Vice-versa, if the charging current exceeds the expected charge current, the PWM value is decremented to decrease the charging current. Charging is immediately stopped if either battery temperature exceeds the maximum threshold or if the battery voltage exceeds 9.2V. This happens due to overheating, respectively because the battery is fully charged. Stopping the charging is done by setting the PWM value to 0 and disabling charging (`isCharging` is set to false).

For this algorithm, the charging current is controlled using a PWM signal generated on GPIO23 (PWM channel 6). The PWM frequency is set to 30kHz (needed for system responsiveness), with an 8-bit resolution. The PWM value is dynamically adjusted based on the measured charging current compared to the expected charge current to maintain a safe and efficient charging process.

During charging, the sensor task's priority is increased to ensure timely adjustments to the PWM signal and accurate sensor readings. Continuous monitoring ensures that charging is halted if unsafe conditions, such as high temperature or over-voltage, are detected.

5.2 MOBILE APPLICATION SUBSYSTEM

The primary objective of the developed mobile application is to offer users a comprehensive system for monitoring battery status and parameters, alongside enabling control over the electric vehicle. To accomplish this, an expo-style project has been initiated using `npx create-expo-app@latest` terminal command [16].

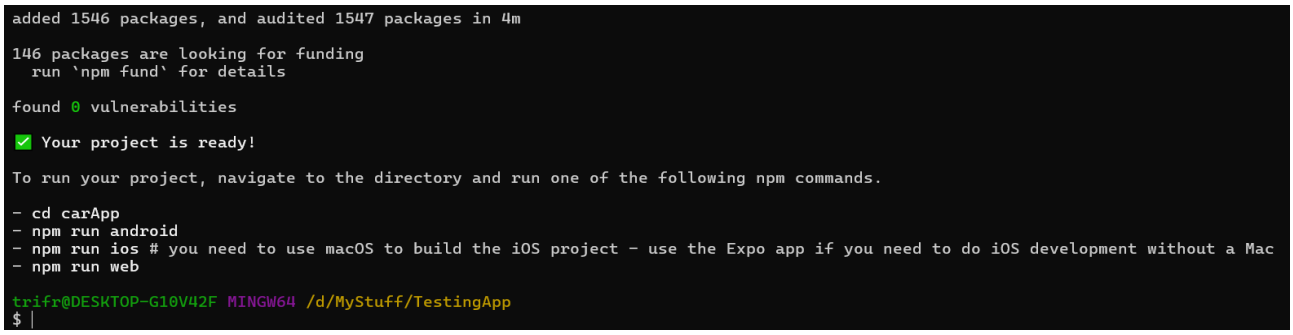

Figure 5.11: "Expo project set-up"

The next step involves creating the necessary screens, their designs and functioning logic. The structure of the documents managing the application's user interface can be divided into:

1. Components directory: contains reusable elements, facilitating easier code reuse and maintaining a clean, well-organized code structure.
2. Navigation directory: includes all the logic for navigation within the application.

3. Screens directory: as detailed in the chapter on the mobile application's architecture, this directory contains the three main screens of the graphical interface: MainScreen, StatusScreen and ControlScreen.

The components directory includes two elements: the `CustomButton` component and the `Joystick` component.

```
return (
  <Pressable
    onPress={onPress}
    style={[
      styles[`container_${type}`],
      backgroundColor,
      marginVertical ? { marginVertical: marginVertical } : { marginVertical: 10 },
      marginHorizontal ? { marginHorizontal: marginHorizontal } : { marginHorizontal: 0 },
      borderColor ? { borderColor: borderColor } : { borderColor: "transparent" },
      width ? { width: width } : { width: 'auto' },
      height ? { height: height } : { height: 'auto' },
    ]}
  >
    <Text style={[styles[`text_${type}`], fgColor ? { color: fgColor } : {}]}>{text}</Text>
  </Pressable>
);
```

Figure 5.12: "Code section for CustomButton component"

The `CustomButton` component is designed for multiple functionalities, which is why it is categorized as a reusable element. This component takes several properties, including `onPress`, `text`, `type`, `fgColor`, `bgColor`, `marginVertical`, `marginHorizontal`, `borderColor`, `width`, and `height`. The `onPress` property is a function that defines what happens when the button is pressed. The `text` property specifies the text to be displayed on the button. The `type` prop determines the style of the button, choosing between primary and secondary styles defined in the styles object.

`Joystick` component provides a touch-controlled interface that allows users to move an on-screen joystick and receive real-time feedback about its position.

```
const panResponder = useRef(
  PanResponder.create({
    onStartShouldSetPanResponder: () => true,
    onMoveShouldSetPanResponder: () => true,
    onPanResponderGrant: () => {
      position.setOffset({
        x: position.x._value,
        y: position.y._value,
      });
      position.setValue({ x: 0, y: 0 });
    },
    onPanResponderMove: Animated.event(
      [null, { dx: position.x, dy: position.y }],
      { useNativeDriver: false,
        listener: (event, gestureState) => handleMove(gestureState) }
    ),
    onPanResponderRelease: () => {
      position.flattenOffset();
      Animated.spring(position, {
        toValue: { x: 0, y: 0 },
        useNativeDriver: false,
      }).start(() => onStop());
    },
  })
).current;
```

Figure 5.13: "Create joystick functionality using PanResponder"

This component utilizes several key aspects of React Native, including `PanResponder` for touch handling and `Animated` for smooth movement and visual feedback. It uses the `useState` hook to create an `Animated.ValueXY` object named `position`, which tracks the joystick's current position. The `maxDistance` constant is calculated based on the `size` prop, representing the maximum distance the joystick can move from its center. The main functions of this behavior are:

- `onStartShouldSetPanResponder` and `onMoveShouldSetPanResponder`: These return `true` to enable touch and move responses.
- `onPanResponderGrant`: Sets the initial position offset and resets the current value, preparing for movement.
- `onPanResponderMove`: Tracks joystick movement using `Animated.event` to map

gesture displacements to the `position` object. The `handleMove` function processes these movements.

- `onPanResponderRelease`: Animates the joystick back to its center using `Animated.spring`.

The navigation directory sets up a navigation system for the mobile application using the `@react-navigation` library. The primary focus of this code is to define and manage the app's navigation stack. First, the code imports necessary modules and components. `useEffect` and `useRef` from React are used for handling side effects and accessing the navigation object, respectively. The `NavigationContainer` and `createStackNavigator` are essential components from `@react-navigation/native` and `@react-navigation/stack`, respectively, to set up the navigation context and stack navigator. `CardStyleInterpolators` are imported to customize the screen transition animations. Additionally, `Notifications` from `expo-notifications` are used to handle incoming notifications.

A stack navigator is created using `createStackNavigator`, with `NavigationContainer` and `navigationRef` to manage navigation context and references. The `useEffect` hook sets up a listener for notification responses to navigate to `StatusScreen` upon receiving data. `NavigationContainer` wraps the stack navigator, with `Stack.Navigator` configuring screen options like hidden headers, gesture-enabled navigation, and custom screen transitions. Each screen is defined within the stack, enabling smooth navigation throughout the app.

In this project, I have employed two primary approaches to connect with Firebase Services: client-side code and backend code. Here's an explanation of what I aimed to achieve with each method. Implementing client-side code allows the mobile application to establish a real-time connection with the Firebase database for reading and writing data. This interaction enables the application to perform specific functions based on the data. For instance, reading data on the client side is used to fetch all battery parameters transmitted by the ESP32 to the Firebase database. After retrieving these values, the graphical interface updates in real-time, providing the user with detailed information about the vehicle's batteries.

```
void connectToFirebase() {
  config.api_key = FIREBASE_AUTH;
  config.database_url = FIREBASE_HOST;
  if (Firebase.signUp(&config, &auth, "", "")) {
    Serial.println("Connection established!");
    signupOK = true;
  } else {
    Serial.printf("%s\n", config.signer.signupError.message.c_str());
  }

  config.token_status_callback = tokenStatusCallback; // see addons/TokenHelper.h

  Firebase.begin(&config, &auth);
  Firebase.reconnectWiFi(true);
}
```

Figure 5.14: “Client-side connection with Firebase”

This code initializes Firebase and sets up a connection to the Firebase Realtime Database. It begins by importing essential functions from the Firebase SDK, specifically `initializeApp`, `getDatabase`, `ref`, and `onValue`. The Firebase configuration object includes keys and URLs necessary for connecting to the Firebase services. The application is then initialized using `initializeApp` with the provided configuration, and a database reference is obtained using `getDatabase`. Finally, the code exports `db`, `ref`, and `onValue` to allow other parts of the React Native application to easily interact with the database, enabling real-time data reading and writing functionalities, such as fetching and displaying battery parameters sent by an ESP32 device.

To establish a connection with the Firebase Realtime Database for backend code purposes, it was necessary to generate a Service Account Key and initialize the Firebase Admin SDK using the created key. The key generation was possible using Firebase Console, going to: *Project Overview > Project settings > Generate a new private key*. After the key was successfully downloaded, inside the terminal the command `npm install firebase-admin` was used.

```
const functions = require("firebase-functions");
const fetch = require("node-fetch");
const admin = require("firebase-admin");

const account = require("./esp32-car-data-firebase-adminsdk-mqj0m-d55b5c39f4.json");

admin.initializeApp({
  credential: admin.credential.cert(account),
  databaseURL: "https://esp32-car-data-default-rtdb.firebaseio.com",
});
```

Figure 5.15: “Server-side connection with Firebase”

This code snippet sets up a Firebase Cloud Function that integrates with the Firebase Realtime Database using the Firebase Admin SDK. It begins by importing the necessary modules: `firebase-functions` for handling cloud functions, `node-fetch` for making HTTP requests, and `firebase-admin` for accessing Firebase services. The service account key, stored in a JSON file, is imported and used to authenticate the Firebase Admin SDK. The `admin.initializeApp` function is called with the service account credentials and the database URL, initializing the Firebase app and establishing a secure connection to the Firebase Realtime Database. This setup allows the server to perform administrative tasks which can be triggered by different events.

Sending notifications is absolutely necessary for this project. To achieve this, I chose to continue using the Firebase suite of services, this time with Firebase Functions. Firebase Functions is a serverless framework that allows you to automatically execute backend code in response to various triggers, such as background events, HTTPS requests, or interactions with the Admin SDK. To integrate Firebase Functions within your React Native project environment, several steps are necessary.

To integrate Firebase Functions for sending notifications in my React Native project, I had to install Firebase CLI globally using `npm` and authenticate it with my Firebase account by running `firebase login`. I had to initialize Firebase in my project directory using `firebase init`, selecting the "Functions" option and following the prompts to complete the setup.

Inside the `index.js` file within the initialized Firebase functions directory, the implementation revolves around handling push notifications for the mobile application. The functions are structured to respond to updates in the Firebase Realtime Database, specifically concerning battery levels and temperature sensors, utilizing promises and HTTPS requests for the execution.

The `batteryLevelNotification` function triggers upon updates to the `/sensors/vBatt` path in the database. It calculates the current battery percentage based on voltage readings and proceeds to send notifications when the battery either reaches full charge (100%) or drops to a low level ($\leq$ 10%). These notifications, crafted using promises, inform users about the status of their vehicle's batteries, ensuring timely recharging.

The process involves making an HTTPS POST request to the Expo push notification endpoint (`https://exp.host/--/api/v2/push/send`). This request includes a JSON payload detailing the notification message, such as title, body, and data related to battery levels. The `fetch` function, integrated with `async/await` syntax, handles this HTTP request asynchronously. Upon receiving a response from Expo's servers, the function parses the JSON response to log relevant information and handles any errors that may occur during the notification sending process.

```
try {
  const response = await fetch("https://exp.host/--/api/v2/push/send", {
    method: "POST",
    headers: {
      "Accept": "application/json",
      "Accept-encoding": "gzip, deflate",
      "Content-Type": "application/json",
    },
    body: JSON.stringify(message),
  });

  const data = await response.json();
  console.log("Expo push notification response:", data);

  if (data.errors) {
    console.error("Expo push notification error:", data.errors);
  }
} catch (error) {
  console.error("Error sending push notification:", error);
}
```

Figure 5.16: "Sending push-notifications using HTTPS POST request"

The `temperatureNotification` function monitors changes in temperature sensor values (`/sensors/tBatt1` and `/sensors/tBatt2`). Upon updates, it determines whether the temperature readings indicate a normal, warning, or critical level based on predefined thresholds. It then uses HTTPS requests to send corresponding push notifications to inform users about potential overheating issues with their vehicle's batteries. Both functions utilize the Firebase Admin SDK for accessing the database (`admin.database().ref(...)`) to fetch the push token (`/token`) and update notification statuses (`/notification`). After the coding part is completed, the deployment of the functions to the Firebase servers was done using `firebase deploy --only functions`.

Receiving and managing notifications in this project uses Expo's push notification service, which requires configuring by installing specific libraries. The request function asks for permission to send notifications on Android. It shows a dialog with options to grant or deny the permission. If denied, an alert informs the user they won't receive notifications. `Notifications.setNotificationHandler` configures how notifications are handled when received, ensuring they show an alert without playing a sound or setting a badge. The `handleRegistrationError` function displays an alert and stops execution if there are errors during registration.

In the App component, the `useEffect` hook registers for push notifications by calling `registerForPushNotificationsAsync` and saving the token to Firebase using `saveTokenToDatabase`. The `registerForPushNotificationsAsync` function:

- Checks and requests notification permissions if needed and sets up a notification channel.
- Ensures the device is physical, requests permissions, retrieves the Expo push token using the project ID and handles errors.

Setting up the environment for push notifications in a React Native app with Expo requires installing `expo-notifications` and `expo-device` because they provide essential functionality for permission requests, notification handling and device interactions.

Once the system functionalities were successfully implemented, the Expo project needed to migrate to the preview stage. This required utilizing the EAS platform, a streamlined service for building and deploying mobile applications provided by Expo. As previously, there are a few steps that need to be accomplished. To start, I configured the EAS CLI in my development environment by globally installing it using npm, granting access to its commands from any directory on my system. After ensuring it was properly set up to interact with my Expo projects, I proceeded to connect my Expo project to EAS with `eas login`. Following the authentication prompts in the terminal, I linked my Expo account to EAS for integrated management.

Preparing for the build phase involved setting up crucial configuration files like `eas.json` within my project directory. These files specified essential build parameters such as the development or preview mode, targeting specifically Android as the platform and any custom configurations specific to my application's requirements. Executing `eas build --profile preview --platform android` initiated the build process, automating the compilation and bundling of my Expo project tailored for previewing on Android devices. This command efficiently packaged the application into a deployable format suitable for testing or further distribution.

When the build process was completed successfully, I accessed the generated artifacts, meaning the APK file, from the EAS dashboard and I downloaded it on my physical device.

All the necessary source code files can be accessed on the GitHub link [17].

6. TESTING

The Battery Management System is a project that highlights the capabilities of a dynamic charging system combined with a well-designed monitoring system. Its purpose is to extend battery life and protect the electric vehicle from potential risk factors, particularly those that could lead to unwanted fire hazards. To implement and develop this project, various hardware and software components were utilized.

After assembling all the components, the next step is testing to verify that everything works just fine together. Ensuring the project meets safety standards and fulfills the proposed specifications involves thorough testing using multiple methods. These methods are based on two well-known concepts: white-box testing and black-box testing. The use of these testing methods aims to ensure that:

- The step-down converter effectively charges the batteries.
- Sensor data is collected and well managed by the realtime database.
- Mobile application benefits of responsiveness and behaves appropriately

To achieve these objectives, unit testing, integration testing and system testing were used. In this context, unit testing targets each hardware component separately, ensuring there are no electrical malfunctions, verifying that sensor readings are accurate, and confirming that the PWM value adjusts correctly based on given load voltages. Similarly, unit testing for the mobile application focuses on individual UI components and the correctness of data handling from Firebase.

Integration testing examines the interactions between hardware components and between the hardware and mobile application, ensuring they work together seamlessly, such as verifying that sensors correctly interface with the ESP32 and that PWM signals properly control the converter, as well as ensuring the app accurately retrieves and displays sensor data.

System testing validates the entire setup in real-world scenarios, ensuring that the mobile application accurately displays sensor data, sends notifications appropriately and maintains reliable communication with the hardware components through the Firebase RTDB.

6.1 HARDWARE COMPONENTS TESTING

In this step, all the separate modules which offer information regarding the battery or the charging process were tested to check that provided data is correct. The following readings were made using Serial Plotter feature from Arduino IDE while the system was active.

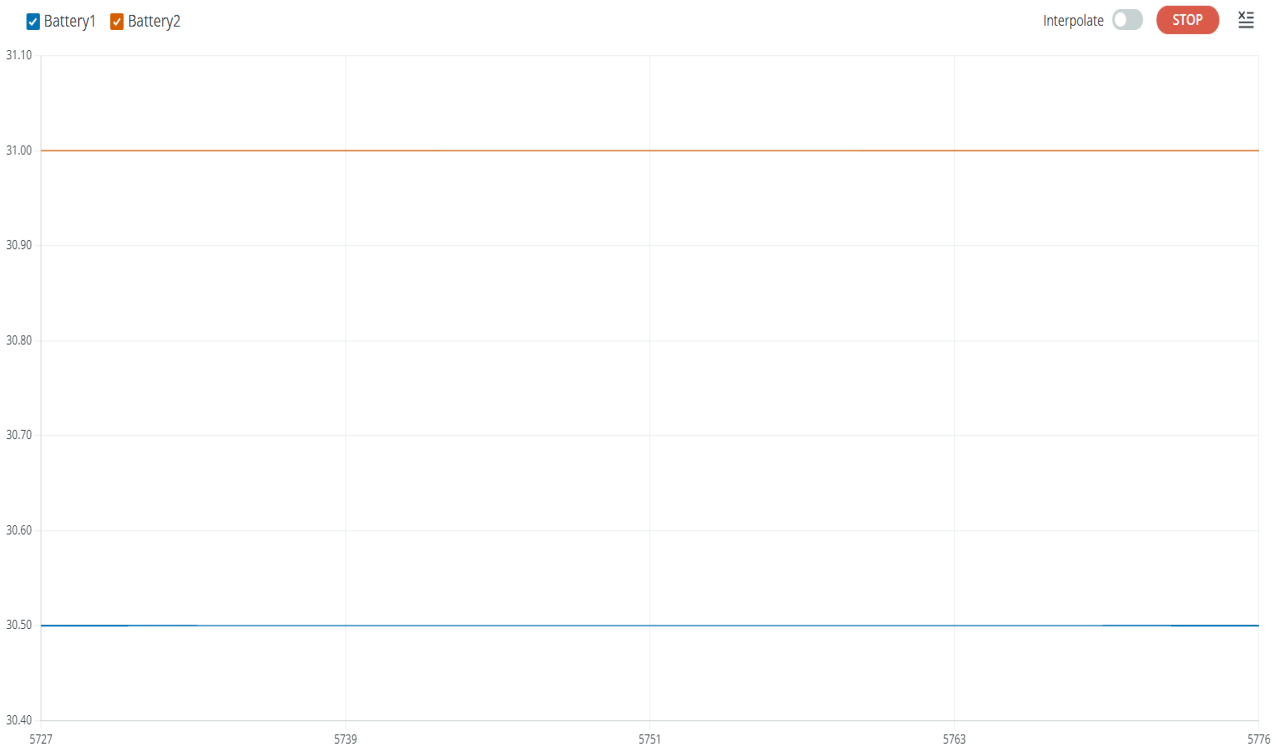

Figure 6.1: “Temperature sensors measurements”

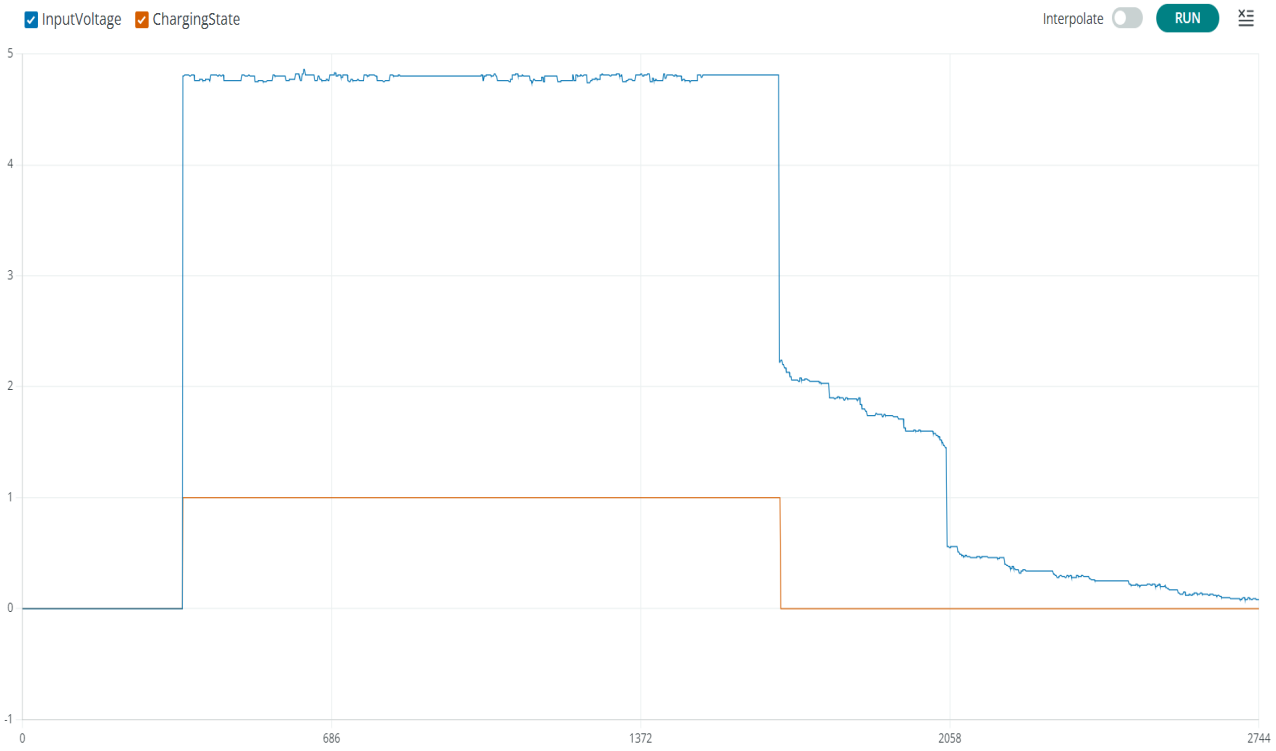

Figure 6.2: “Voltage divider readings”

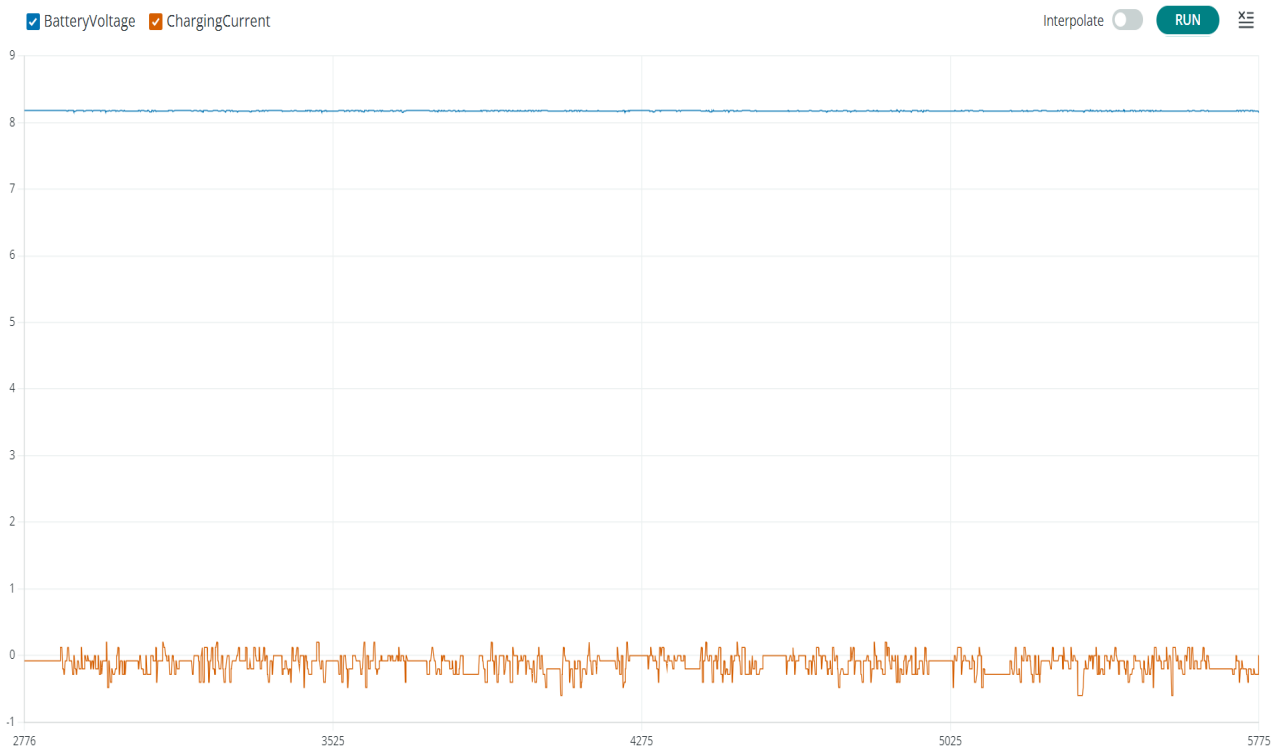

Figure 6.3: “Current sensor measurements”

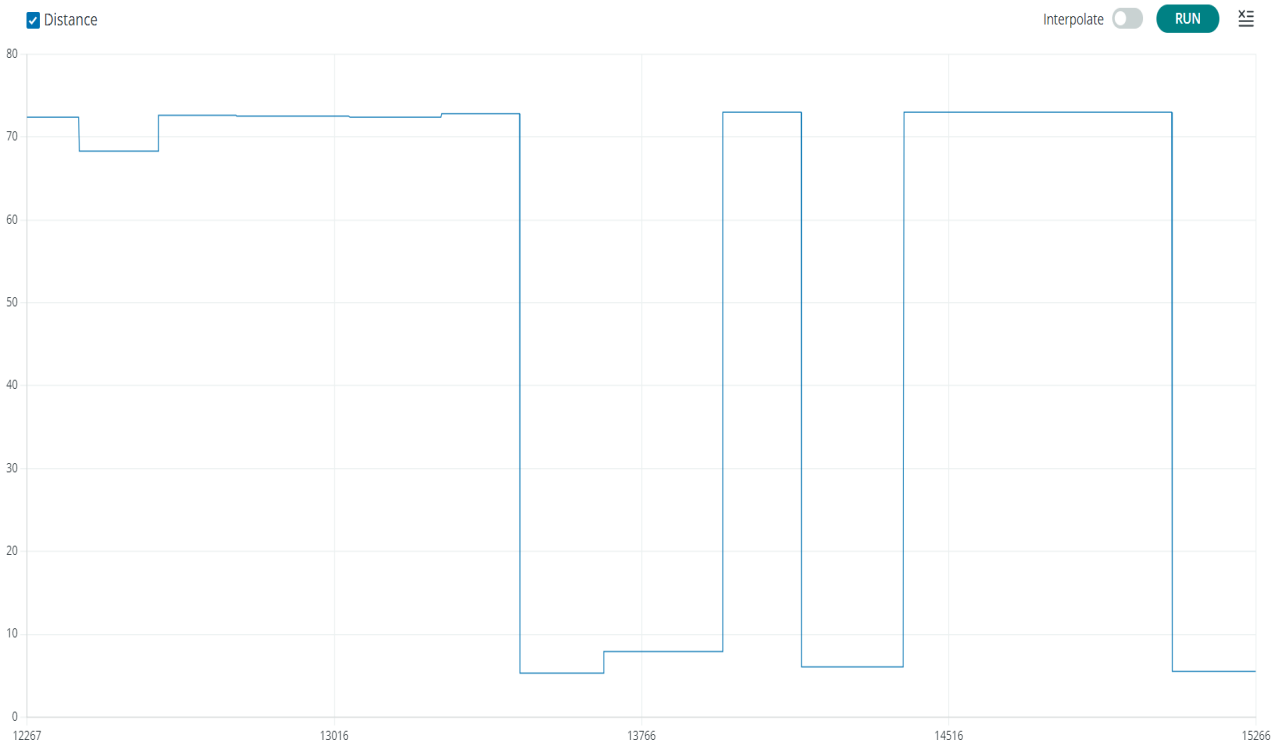

Figure 6.4: “Ultrasonic sensor measurements”

The results obtained after performing the tests conclude the following:

- Most of the time, both batteries will have similar temperatures since they are in the same environment. A significant temperature difference in one of the batteries could signal a

problem that needs attention.

- Figure 6.2 illustrates the initiation of the charging system when the converter is connected to a power source (approximately 5 volts in this example). Subsequently, the system is disconnected to observe its return to the initial state.
- To simulate the presence of an object near the ultrasonic sensor, several cycles of bringing the object close and then removing it were made. Figure 6.4 exposes the results.

To demonstrate the utility of the project and the correct functioning of the system, the following graph was created. It clearly underlines how the PWM signal responds based on the feedback received from the current sensor and maintains the voltage at 5V (a predetermined random value), meaning that charging can be adjusted according to the battery's needs.

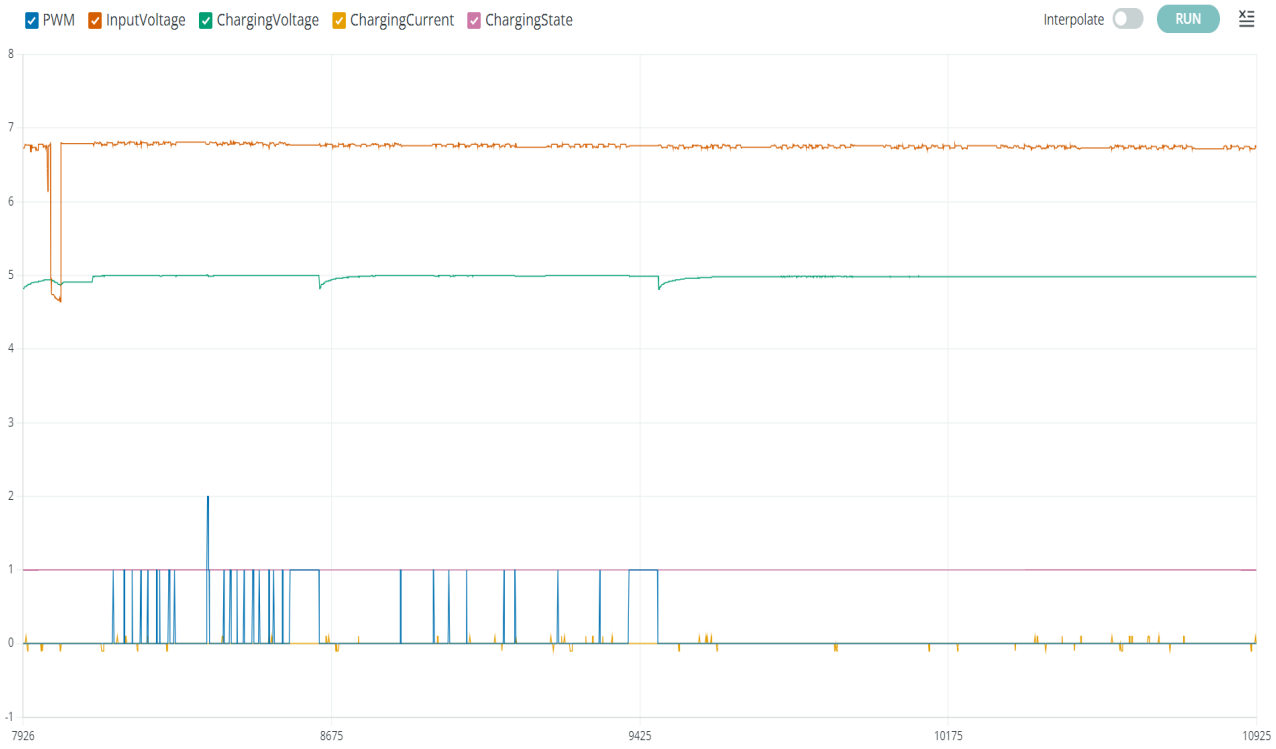

Figure 6.5: "System's behaviour in normal conditions"

Another simulated scenario is when the battery temperature exceeds the maximum accepted limit. Figure 6.6 shows how the system changes its state after the battery temperature surpasses 40 degrees. Consequently, the charging process stops and the voltage level drops.

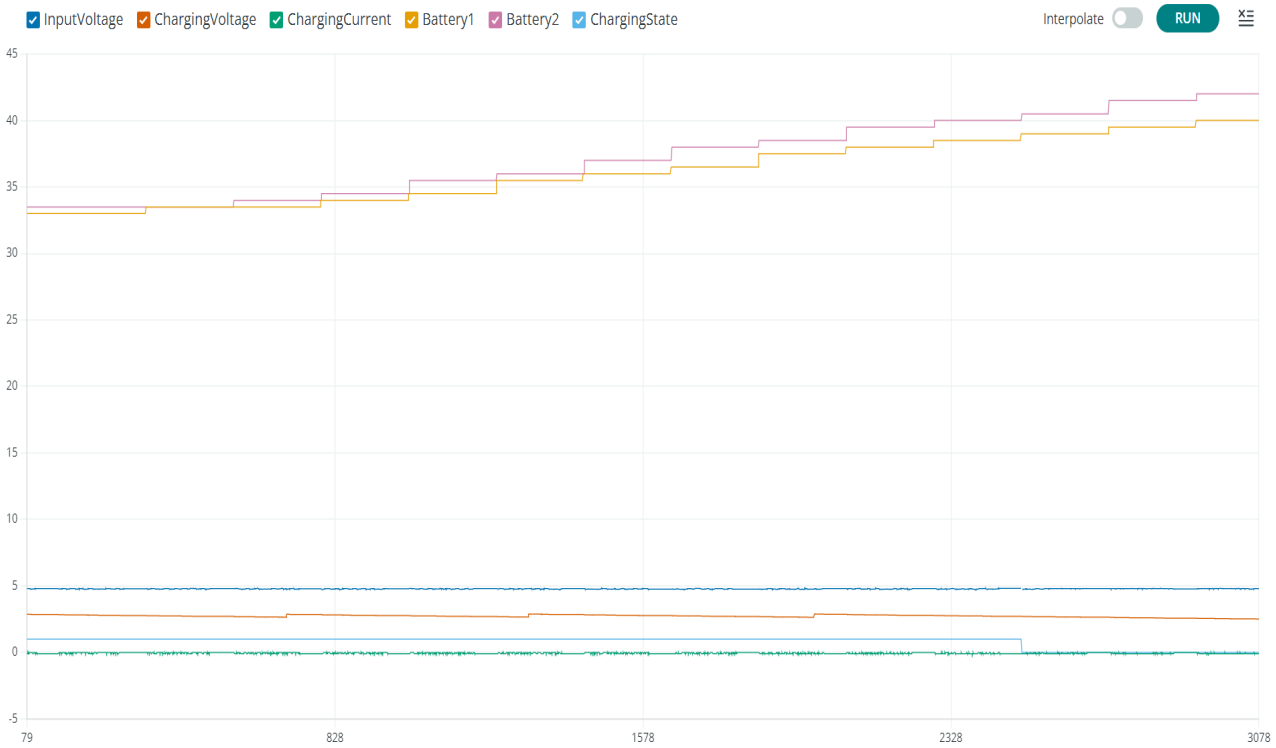

Figure 6.6: “System’s behaviour when the temperature increases”

6.2 FIREBASE AND MOBILE APPLICATION TESTING

A simple way to test the functionality of the mobile application and the monitoring system is by hard-coding the RTDB to simulate the parameters that trigger the events we want to observe in the application. Using the Firebase console web interface, the following results were obtained:

- Fully charged battery

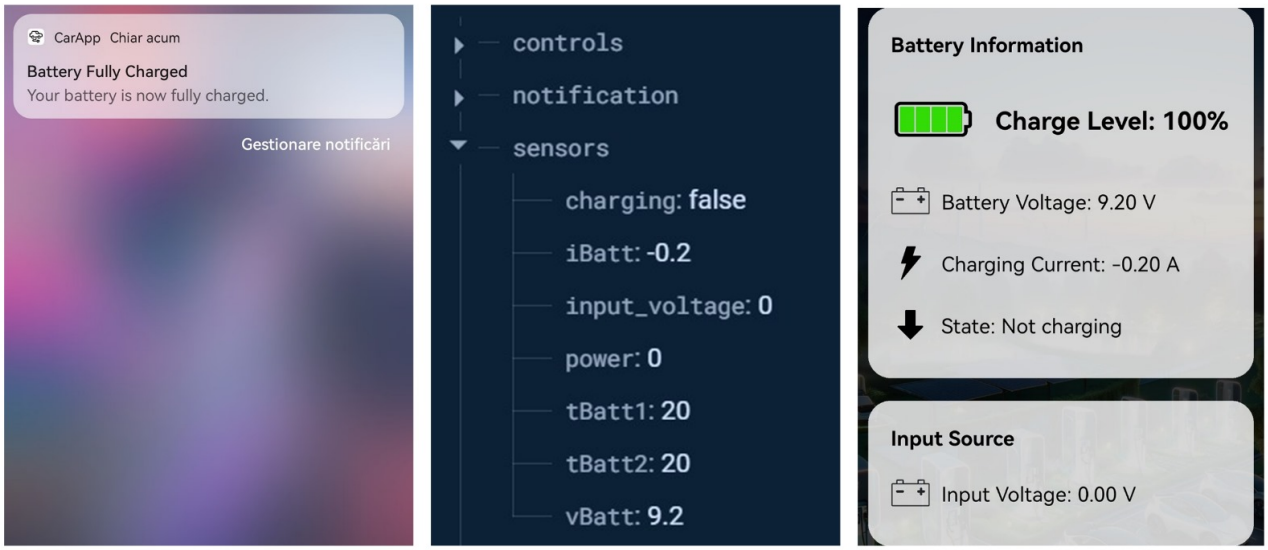

Figure 6.7: “Tests when battery is fully charged”

- Battery level $\leq$ 10%

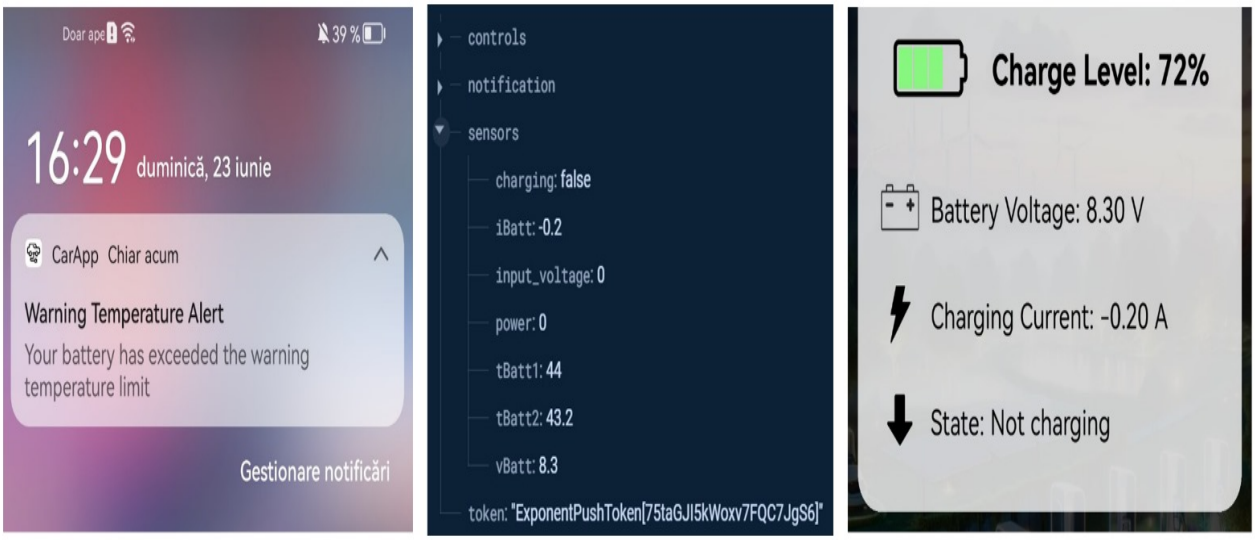

Figure 6.8: “Tests when battery is overheating”

- Battery overheat

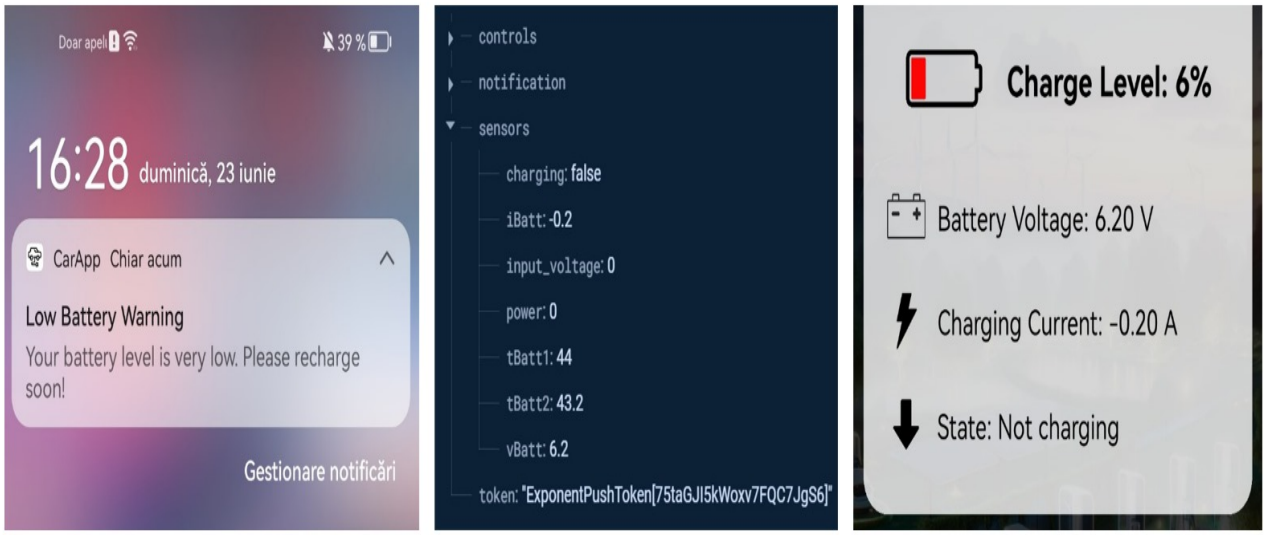

Figure 6.9: “Tests when battery is close to discharge”

After executing the tests, it was concluded that the Battery Management System performed as expected in terms of reliability and responsiveness to various conditions.

7. CONCLUSIONS

7.1 PROJECT OVERVIEW

The development of a Battery Management System (BMS) for an electric vehicle has been a profound and challenging journey, combining relevant speciality domains such engineering and programming. This project successfully demonstrated its abilities to charge safely, to monitor and to control a vehicle. Reflecting on the broader context, this project aligns with the growing movement towards sustainable transportation solutions. By focusing on maximizing battery lifespan and improving overall autonomy and safety, this project contributes to the ongoing efforts to make electric vehicles more viable and sustainable.

On a personal note, this project marks a significant milestone in my academic and professional journey. It lead me into learning and enriching my know-ledge into speciality literature of cars and electronics. My passion for cars, technology and software development has driven me to explore and create this Battery Management System which conclude to be a real success.

7.2 FUTURE WORK

One initial idea for improving the project is to enhance the charging system. Based on the concluded tests and measurements, I have identified minor improvements that can be made by optimizing the components I used. These changes can increase the efficiency of the charging process and improve the converter's performance. For instance, most converters used in electric vehicles employ advanced charging techniques. In my project, I could replace the buck converter diode with another MOSFET to achieve better results.

Another aspect I aim to improve is the implementation of a machine learning algorithm. This algorithm would automate the collection of data from the RTDB, using it to train and test a learning model to predict battery life expectancy. This algorithm could be deployed as an API and integrated into the mobile application. Such an enhancement would be valuable for my next professional step, which is enrolling in a master's program.

Currently, the application is not designed to operate at a production scale for multiple users. However, this is a potential area for improvement. Implementing an authentication and login system would be one of the solutions to support multiple users.

8. BIBLIOGRAPHY

[1] ***, "Commission regulation (eu) no 459/2012 of 29 may 2012 amending regulation (ec) no 715/2007 of the european parliament and of the council and commission regulation (ec) no 692/2008 as regards emissions from light passenger and commercial vehicles (euro 6). off. j. eur. union 2012, 48, 258–266," 2012, Accessed on June 3, 2024. [Online]. Available: https://eur-lex.europa.eu/eli/reg/2012/459/oj#.

[2] A. Dorsz and M. Lewandowski, "Analysis of fire hazards associated with the operation of electric vehicles in enclosed structures," *Energies*, vol. 15, no. 1, p. 11, Dec. 2021, Accessed on June 22, 2024, ISSN: 1996-1073. DOI: 10.3390/en15010011. [Online]. Available: http://dx.doi.org/10.3390/en15010011.

[3] C. D. Koolen and G. Rothenberg, "Air pollution in europe," *ChemSusChem*, vol. 12, no. 1, 164–172, Dec. 2018, Accessed on June 22, 2024, ISSN: 1864-564X. DOI: 10.1002/cssc.201802292. [Online]. Available: http://dx.doi.org/10.1002/cssc.201802292.

[4] I. Publications, "Global ev outlook 2021," vol. 20, no. 1, pp. 63–73, 2021, Accessed on June 3, 2024. [Online]. Available: https://www.iea.org/reports/global-ev-outlook-2021.

[5] D. Ronanki, A. Kelkar, and S. S. Williamson, "Extreme fast charging technology—prospects to enhance sustainable electric transportation," *Energies*, vol. 12, no. 19, p. 3721, 2019, Accessed on June 22, 2024, ISSN: 1996-1073. DOI: 10.3390/en12193721. [Online]. Available: http://dx.doi.org/10.3390/en12193721.

[6] Espressif, *Esp32-devkitc v4 getting started guide*, https://docs.espressif.com/projects/esp-idf/en/latest/esp32/hw-reference/esp32/get-started-devkitc.html, Accessed on June 22, 2024.

[7] T. Instruments, *Zerø-drift, bi-directional current/power monitor with i2c™ interface*, https://cdn-shop.adafruit.com/datasheets/ina219.pdf, Accessed on June 22, 2024.

[8] ST, *L298 dual full-bridge driver*, https://www.sparkfun.com/datasheets/Robotics/L298$_{HB}ridge.pdf$, Accessed on June 22, 2024.

[9] M. Fetcenko, S. Ovshinsky, B. Reichman, *et al.*, "Recent advances in nimh battery technology," *Journal of Power Sources*, vol. 165, no. 2, 544–551, Mar. 2007, Accessed on June 22, 2024, ISSN: 0378-7753. DOI: 10.1016/j.jpowsour.2006.10.036. [Online]. Available: http://dx.doi.org/10.1016/j.jpowsour.2006.10.036.

[10] D. Nayanasiri and Y. Li, "Step-down dc–dc converters: An overview and outlook," *Electronics*, vol. 11, no. 11, p. 1693, May 2022, Accessed on June 22, 2024, ISSN: 2079-9292. DOI: 10.3390/electronics11111693. [Online]. Available: http://dx.doi.org/10.3390/electronics11111693.

[11] H. Hutri, “Comparison of react native and expo,” 2023, Accessed on June 22, 2024. [Online]. Available: `https://lutpub.lut.fi/handle/10024/165256`.

[12] L. S. Vailshery, *Cross-platform mobile frameworks used by software developers worldwide from 2019 to 2023*, https://www.statista.com/statistics/869224/worldwide-software-developer-working-hours/, Accessed on June 22, 2024.

[13] Firebase, *Developer documentation for firebase*, https://firebase.google.com/docs, Accessed on June 22, 2024.

[14] S. Cuibus, *Chassis design implmentation*, Accessed on June 22, 2024.

[15] Expo, *Send notifications with expo’s push api*, https://docs.expo.dev/push-notifications/sending-notifications/, Accessed on June 22, 2024.

[16] Expo, *Create amazing apps that run everywhere*, https://docs.expo.dev/, Accessed on June 22, 2024.

[17] T. Robert-Nicolas, *Github repository of the project: Battery management system for small electric vehicle*, https://github.com/trifrobert/Licenta.git, Accessed on June 22, 2024.

STATEMENT REGARDING
THE AUTHENTICITY OF THE THESIS PAPER *

I, the undersigned TRIF ROBERT - NICOLAS,
Identifying myself with CI series AR no. 929612,
CNP (personal numerical code) 5011117020087
author of the thesis paper BATTERY MANAGEMENT SYSTEM FOR SMALL ELECTRIC VEHICLE
Developed with the purpose of participating in the graduation examination completing the educational level of BACHELOR DEGREE organized by the Faculty of COMPUTER SCIENCE within the Politehnica University of Timişoara, session JUNE 2024 of the academic year 2023-2024, coordonated by CONF. DR. ING RĂZVAN BOGDAN, considering Article 34 of the *Regulation on the organization and conduct of bachelor/diploma and dissertation examinations*, approved by Senate Decision no. 109/14.05.2020, and knowing that in the event of subsequent finding of false statements, I will bear the administrative sanction provided by Art. 146 of Law no. 1/2011 – law of national education, namely the cancellation of the diploma of studies, I declare on my own responsibility that:

- This paper is the result of my own intellectual endeavour,
- He paper does not contain texts, data or graphic elements taken from other papers or from other sources without such authors or sources being quoted, including when the source is another paper / other works of my own;
- bibliographic sources have been used in compliance with Romanian legislation and international copyright conventions.
- this paper has not been publicly presented, published or presented before another the bachelor/diploma/dissertation examination committee.
- In the development of the paper I have used instruments specific for artificial intelligence (AI), namely CHATGPT (name) OPEN AI (source), which I have quoted within the paper / I have not used instruments specific for artificial intelligence (AI)[1].

I declare that I agree that the paper should be verified by any legal means in order to confirm its originality, and I consent to the introduction of its content in a database for this purpose.

Timişoara,
Date 21.06.2024

Signature

* The statement will be filled-in by the student, will be signed by hand by them and inserted at the end of the thesis paper, as a part of it.
[1] One of the variants will be selected and inserted in the statement: 1 – AI has been used, and the source will be mentioned, 2 – AI has not been used