BINDURA UNIVERSITY OF SCIENCE EDUCATION (BUSE)

DEPARTMENT OF ELECTRONIC ENGINEERING AND PHYSICS

FINAL YEAR PROJECT

PROJECT TITLE:

SMART IOT-BASED BATTERY MANAGEMENT SYSTEM WITH TRACKING SYSTEM AND TERMINAL SECURITY

NAME(S) OF RESEARCHER(S)

JOE PANASHE NYIKA

SUPERVISOR: ENG. KOMICHI

YEAR: 2024 - 2025

DECLARATION

This research project is my original work except where sources have been acknowledged . The work has never been submitted, nor will it ever be, to another university to award a degree or any other qualification.

STUDENT NAME: JOE PANASHE NYIKA

SIGNATURE:…….

SUPERVISOR NAME: ENGINEER S. KOMICHI

SIGNATURE : ……

ACKNOWLEDGEMENTS

Firstly, I am grateful to The Almighty God for giving me the strength, capability, and ability to complete this project. Furthermore, this project would not have been successful without support and guidance from my supervisor Engineer S. Komichi. I would also acknowledge all lecturers at large in the Electronic Engineering and Physics Department at the Bindura University of Science education. I appreciate their prayers for my success.

ABSTRACT

This project presents the design and implementation of an intelligent, IoT-enabled Battery Management System (BMS) that ensures safe, efficient, and secure battery operation through real-time monitoring and control. The system is built around the ESP32 microcontroller, which provides dual-core processing capabilities, integrated Wi-Fi and Bluetooth communication, and support for multiple peripheral interfaces. Core components include the ACS712 current sensor for accurate DC current measurement, a 25V voltage sensor for real-time voltage monitoring, and the DS18B20 digital temperature sensor for precise thermal data acquisition. These sensors feed into the ESP32, which processes the data to detect critical conditions such as overcharging, deep discharging, and thermal runaway, while also performing real-time cell balancing and temperature regulation.

A secure authentication mechanism using a 4x4 matrix keypad restricts access to battery terminals, ensuring protection against unauthorized use. The NEO-6M GPS module enables accurate tracking of the battery's geographical location, contributing to theft prevention and enhanced asset management. The system interfaces with an I2C-based LCD for local parameter display and employs status LEDs to indicate battery health. Furthermore, the ESP32 transmits collected data to a user-friendly HTML-based webpage and a hybrid Android application developed using the Ionic framework. The mobile app displays battery voltage, current, and temperature in real-time via graphical gauges and plots battery level history alongside a dynamic location map using Leaflet.js.

A MySQL database managed through XAMPP stores all measured parameters for analysis, fault detection, and historical tracking. Proteus simulation was employed during system validation to test circuit integrity and sensor interactions in a virtual environment before physical deployment. The resulting prototype provides a low-cost, scalable, and modular solution applicable in electric vehicles, solar energy storage systems, and other battery-powered infrastructure, significantly enhancing battery safety, performance, and lifespan through intelligent monitoring and remote accessibility.

TABLE OF CONTENTS

TABLE OF FIGURES

LIST OF TABLES

CHAPTER 1: INTRODUCTION

1.0 Background of study

Battery management systems (BMS) are critical components in ensuring the efficiency, safety, and longevity of battery-powered systems. The development of BMS has evolved significantly over the years, driven by advancements in technology and the increasing demand for reliable and efficient energy storage solutions. The earliest battery management systems were relatively simple, focusing primarily on basic functions such as monitoring voltage and current to prevent overcharging and deep discharging. However, as battery technology advanced, particularly with the introduction of lithium-ion batteries, the complexity and capabilities of BMS also increased. Modern BMS now incorporate sophisticated algorithms and real-time monitoring of multiple parameters, including temperature, state of charge (SoC), and state of health (SoH). A typical BMS consists of several key components, including sensors for voltage, current, and temperature, a microcontroller for data processing, and communication modules for interfacing with external systems. The primary functions of a BMS include monitoring and control, protection, balancing, and data logging and communication.

The integration of real-time tracking and terminal security features into BMS represents a significant advancement in battery management technology. Real-time tracking, typically achieved through GPS modules, allows for precise monitoring of the battery's location, which is crucial for asset management and theft prevention. Terminal security involves implementing data encryption, authentication mechanisms, and other cybersecurity measures to protect against unauthorized access and tampering. Battery management systems are widely used in various applications, including electric vehicles (EVs), renewable energy systems, and portable electronic devices. The future of BMS is likely to see further integration of advanced technologies such as machine learning and artificial intelligence to enhance predictive maintenance and optimize battery performance. Additionally, the development of cloud-based BMS (CBMS) offers higher computational resources and improved performance through advanced digital twin models and algorithms. In conclusion, the evolution of battery management systems has been driven by the need for more efficient, safe, and reliable energy storage solutions. The integration of real-time tracking and terminal security features represents a

significant step forward in enhancing the capabilities of BMS. As technology continues to advance, the future of BMS looks promising, with potential applications in a wide range of industries and ongoing improvements in performance and security.

1.1 Problem statement

Modern battery-powered systems require comprehensive real-time monitoring to ensure optimal performance, safety, and increase battery life The absence of integrated solutions for tracking key parameters like current, voltage, and temperature increases the risk of overcharging, deep discharging, and thermal runaway, leading to potential battery failure and safety hazards. Additionally, the lack of real-time location tracking complicates asset management and theft prevention. Existing solutions often fail to address these issues holistically, lacking secure communication and user-friendly interfaces.

1.2 Problem Solution

Developing an integrated battery management system (BMS) with real-time monitoring, GPS based location tracking, and advanced security features which uses sensors to continuously monitor critical battery parameters which include battery voltage, current, temperature and charging status and incorporates a GPS module for location of battery.

1.3 Aim

To design and develop a smart battery management system capable of real-time monitoring of battery parameters which are current, voltage and temperature and implements a GPS based location tracking

1.4 Objectives

1. To develop a robust system that continuously monitors battery current, voltage, and temperature to autonomously detect critical conditions in real-time.
2. To integrate a GPS module within the battery management system, providing real-time tracking of the battery's location.
3. To integrate terminal security features to protect against unauthorized terminal access and data breaches.
4. To design and develop an intuitive and user-friendly Android application that displays real-time battery voltage, current, temperature and location data.

1.5 Conclusion

In conclusion, the development of an integrated battery management system (BMS) that combines real-time monitoring, GPS based location tracking, and advanced security features addresses the critical challenges faced by modern battery-powered systems. By continuously monitoring key battery parameters, ensuring precise location tracking, and implementing robust security measures, this system enhances operational efficiency, safety, and asset management. The user-friendly Android application further facilitates efficient monitoring and management, offering a comprehensive solution that meets the growing demand for reliable and advanced battery management technologies.

CHAPTER 2: LITERATURE REVIEW

2.0 Introduction

The literature review constitutes a vital component of the Battery Management System (BMS) research project by offering a comprehensive analysis of existing studies, theoretical frameworks, and technological advancements relevant to the topic. This chapter establishes a foundational understanding of current developments in battery management systems, real-time monitoring, GPS tracking, and terminal security. Through a critical synthesis of existing literature, the discussion identifies key research gaps, unresolved challenges, and the contextual relevance of the proposed system within the broader technological landscape.

Significant evolution has characterized battery management systems over recent years, largely influenced by rapid advancements in battery technologies and the increasing demand for efficient and reliable energy storage solutions. Contemporary BMS platforms incorporate advanced algorithms and real-time data acquisition capabilities to enhance system performance, ensure operational safety, and prolong battery life. The literature covers a wide range of topics, including the historical development of BMS, core components and functionalities, and comparative evaluations of different system architectures and strategies.

The findings of this review will directly inform the design and development of the proposed integrated BMS, with the objective of overcoming limitations observed in existing solutions and aligning with emerging technological demands. The chapter is organized to address each major theme in detail, culminating in a summary that synthesizes key insights and identifies areas requiring further investigation.

2.1 Related Work

Battery Management Systems (BMS)

Battery Management Systems (BMS) are critical for ensuring the safety, efficiency, and longevity of battery packs, especially in applications like electric vehicles (EVs), renewable energy storage, and consumer electronics. The primary objectives of a BMS include monitoring and data collection, ensuring functional safety, cell balancing, thermal management, and extending battery life.

Battery Management Systems (BMS) have become essential components in modern electrical systems, especially with the increasing adoption of energy storage solutions across various sectors. The global shift in electricity usage and distribution has intensified the demand for

Energy Storage Systems (ESS), positioning them among the fastest-growing products in the power sector. Gabbar et al. [1] conducted a comprehensive review on the development of BMS technologies and the evolution of relevant industrial standards. Their study emphasized the fundamental role of BMS in monitoring, controlling, and optimizing the performance of individual or multiple battery modules, particularly in scenarios where safe operation is critical. A key insight from the study is the BMS's dual responsibility to respond to both external and internal events in order to ensure safety and reliability. The authors described BMS as an indispensable management scheme that not only protects batteries but also enhances performance through regulated charging and disconnection mechanisms during abnormal operating conditions. The review further explores BMS applications in electric transportation and stationary energy storage systems, highlighting components such as architecture, system topology, operational logic, and safety functionalities. In addition, the study examines existing industrial standards and regulatory codes governing BMS design and operation. It provides detailed recommendations to bridge gaps in current safety and performance requirements, focusing specifically on the needs of electric transportation and stationary applications. The work by Gabbar et al. significantly contributes to the understanding of how standardized BMS architectures can support the reliable and efficient deployment of large-scale energy storage systems [1].

Sanino [2] provides an introductory yet informative overview of Battery Management Systems, particularly in relation to lithium-ion (Li-ion) battery technology. With energy densities reaching up to 265 Wh/kg, Li-ion batteries offer a high-performance solution for various modern applications. However, the article highlights a major concern: under certain stress conditions, such as overcharging, overheating, or mechanical damage Li-ion batteries may become unstable, posing risks of thermal runaway, combustion, or even explosion. The author emphasizes that BMSs are not just useful, but necessary, for managing these batteries safely. By monitoring vital parameters such as voltage, current, and temperature, BMSs help prevent conditions that could compromise safety or battery lifespan. The article introduces the foundational concepts of BMS design, including sensing units, control circuits, and protection features. Sanino further outlines basic BMS components such as voltage monitors, temperature sensors, state-of-charge estimators, and control logic systems that collectively ensure the battery pack operates within safe limits.

Although the article provides only a surface-level exploration, it successfully contextualizes the importance of BMS in high-energy applications and serves as a gateway for further study into more advanced features such as communication protocols, thermal modelling, and battery health diagnostics. Sanino's contribution is particularly useful for understanding why safety-critical designs are indispensable when deploying Li-ion batteries in high-demand systems [2].

Several academic institutions have contributed to the development of Battery Management Systems (BMS) through experimental and prototype-based projects. A notable project conducted by the Department of Electrical Engineering at Sebelas Maret University in Indonesia focused on designing a BMS for lithium-ion batteries, specifically targeting challenges such as overcharging and excessive temperature rise [3]. The system employed an Arduino Nano microcontroller as its processing unit, supported by basic protection and control circuitry. While the system effectively addressed core safety functions, it lacked advanced features such as real-time remote monitoring or communication interfaces, thereby limiting its applicability in modern networked environments.

In a separate initiative, the Department of Electrical and Electronics Engineering at Vignan Institute of Technology and Science, Telangana, India, developed a BMS tailored for electric vehicle (EV) applications [4]. Their system emphasized the enhancement of safety and reliability by incorporating functionalities such as state-of-charge (SOC) monitoring, temperature regulation, cell balancing, and energy recovery management through regenerative braking. A key innovation in this project was the use of Kalman filtering for SOC estimation, a method recognized for its superior performance in dynamic EV environments compared to other estimation techniques. The application of Kalman filters enabled more accurate and responsive SOC predictions, contributing to better battery utilization and extended lifespan. Both projects underscore the significance of tailoring BMS design to specific use cases stationary vs. mobile and highlight the evolving nature of BMS functionality. However, limitations such as lack of remote diagnostics or cell-level granularity were evident, pointing to opportunities for future system enhancements.

While several prior BMS implementations focused on overall battery protection, they often overlooked cell-level management, an essential aspect for ensuring precision safety and optimal battery health. This limitation was addressed in the work carried out by Orion Company, which developed a lithium-ion BMS specifically designed for electric vehicles and implemented a Controller Area Network (CAN-bus) communication protocol [5]. This system

was capable of sampling real-time battery parameters, estimating the state of charge (SOC)**,** and exchanging data with the vehicle controller. Despite the absence of a cell balancing feature, the BMS was successfully deployed in an electric vehicle manufactured by Tianjin, demonstrating its viability and robustness in real-world applications.

In another Orion Company project, the BMS was designed to manage lithium battery packs composed of multiple individual cells [6]. The system featured continuous cell-level voltage monitoring and advanced current control algorithms. By analysing the health and charge status of each individual cell, the system calculated the maximum allowable charge (source current) and discharge (load current) to prevent cell overvoltage and under voltage conditions. These current limits were dynamically communicated to external devices such as battery chargers, motor controllers, and inverters, thus ensuring the safety and performance of the entire battery system. The Orion BMS also supported thermal protection mechanisms and advanced diagnostics, making it suitable for applications requiring high reliability and long-term battery preservation.

These projects from Orion Company illustrate the importance of integrating fine-grained monitoring and intelligent current management in modern BMS architectures. Their emphasis on cell-level diagnostics and communication with external control systems reflects a significant advancement over basic protection-based systems, making such solutions more adaptable to next-generation electric vehicles and energy systems.

2.2 Limitations of related work

Existing battery management systems (BMS) face several limitations that my proposed Battery Management System aims to address:

- Lack of Real-Time Monitoring: Many BMS solutions do not provide continuous, real-time monitoring of critical battery parameters such as current, voltage, and temperature. This can lead to delayed detection of issues like overcharging, deep discharging, and thermal runaway, which can compromise battery safety and longevity.
- Inadequate Location Tracking: Traditional BMS often lack integrated GPS-based location tracking, making it difficult to manage and secure battery assets. Without precise location data, asset management becomes less efficient, and the risk of theft increases.

- Insufficient Security Measures: Some BMS solutions do not incorporate robust security features, leaving them vulnerable to unauthorized access and tampering. This can result in data breaches and compromised battery performance.
- Complex and Non-User-Friendly Interfaces: Existing BMS interfaces can be complex and difficult to navigate, making it challenging for users to monitor and manage battery systems effectively. A user-friendly interface is essential for ensuring efficient operation and quick response to any issues.

2.3 Theoretical Framework and Design of System Components

2.3.1 Current Measurement

In this project ACS712 is going to be used which lies in the hall effect sensors category. It is being used due to its characteristics which are accuracy, high speed, better isolating function, strong overload ability and no loss of energy.

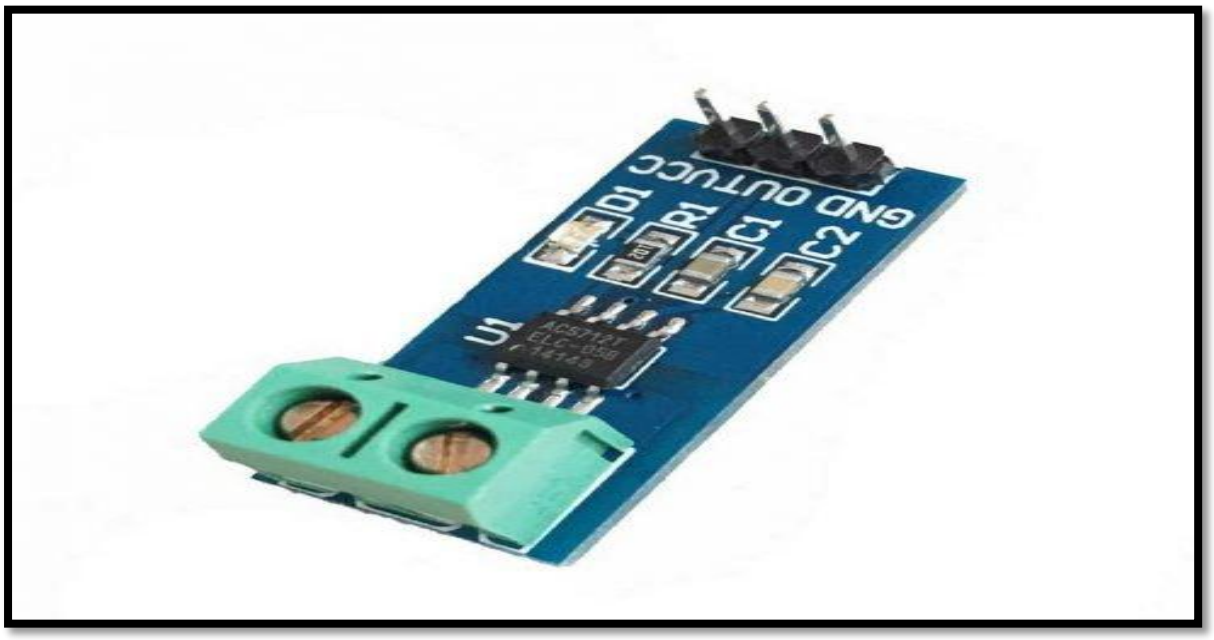


Figure 2.0: ACS712 Current Sensor

Key Features

The current sensor ACS712, Figure 2.0 is a highly integrated Hall-effect-based linear current sensor designed to measure both AC and DC currents. It is available in different versions with current measurement ranges of ±5A, ±20A, and ±30A. The sensor's output voltage is proportional to the measured current, with sensitivities of 185mV/A for the ±5A version, 100mV/A for the ±20A version, and 66mV/A for the ±30A version.

One of its notable features is the 2.1kVRMS voltage isolation, which ensures safety and reliability in various applications. Additionally, the ACS712 features a low-noise analog signal path, ensuring accurate and stable current measurements, and a bandwidth of 80kHz, making it suitable for a wide range of applications. The internal conductor resistance of 1.2mΩ minimizes power loss, and the total output error is 1.5% at 25°C, ensuring precise measurements. The sensor employs the Hall effect principle, where the current flowing through its internal conductor generates a magnetic field detected by the Hall effect sensor. This indirect sensing method provides reliable current measurements without directly measuring the voltage drop. The ACS712 outputs an analog voltage, which is easy to interface with microcontrollers and other analog input devices. It includes a filter pin that allows users to set the bandwidth and reduce noise in the signal. The ACS712 is widely used in motor control circuits, electrical load detection, switched-mode power supplies (SMPS), and over-current protection, making it a versatile and essential component for our BMS.

2.3.2 Temperature measurement

Chemical reactions inside the battery accelerate as the battery's temperature rises. Lithium ion batteries can easily get affected from higher temperatures in a number of ways, including reduced performance and storage capacity. As a result, it must be checked and maintained in order to achieve the optimum results.

Figure 1.1: DS18B20 Temperature Sensor

Figure 2.1 shows a DS18B20 which is a digital temperature sensor that operates on the 1-Wire protocol, allowing it to communicate with a microcontroller using a single data line. It measures temperature in the range of -55°C to +125°C with an accuracy of ±0.5°C. The sensor provides 9-bit to 12-bit resolution, with the default resolution being 12-bit, which translates to a precision of 0.0625°C. The DS18B20 is available in various packages, including SOP, To-92, and even as a waterproof sensor, making it suitable for diverse applications.

Reasons for Using DS18B20 in BMS:

- High Accuracy: The DS18B20 provides precise temperature measurements, which is crucial for monitoring battery health and preventing thermal runaway.
- Single-Wire Communication: Its 1-Wire protocol simplifies wiring and reduces the number of pins required on the microcontroller, making the system more efficient.
- Wide Temperature Range: The sensor's ability to measure temperatures from -55°C to +125°C ensures it can operate effectively in various environmental conditions.
- Low Power Consumption: The DS18B20 draws minimal power, which is beneficial for battery-operated systems.
- Unique Addressing: Each DS18B20 sensor has a unique 64-bit serial code, allowing multiple sensors to be connected to the same data line without address conflicts.
- Alarm Functionality: The sensor can be programmed with user-defined temperature thresholds to trigger alarms, enhancing safety measures.

2.3.3 Voltage Measurement

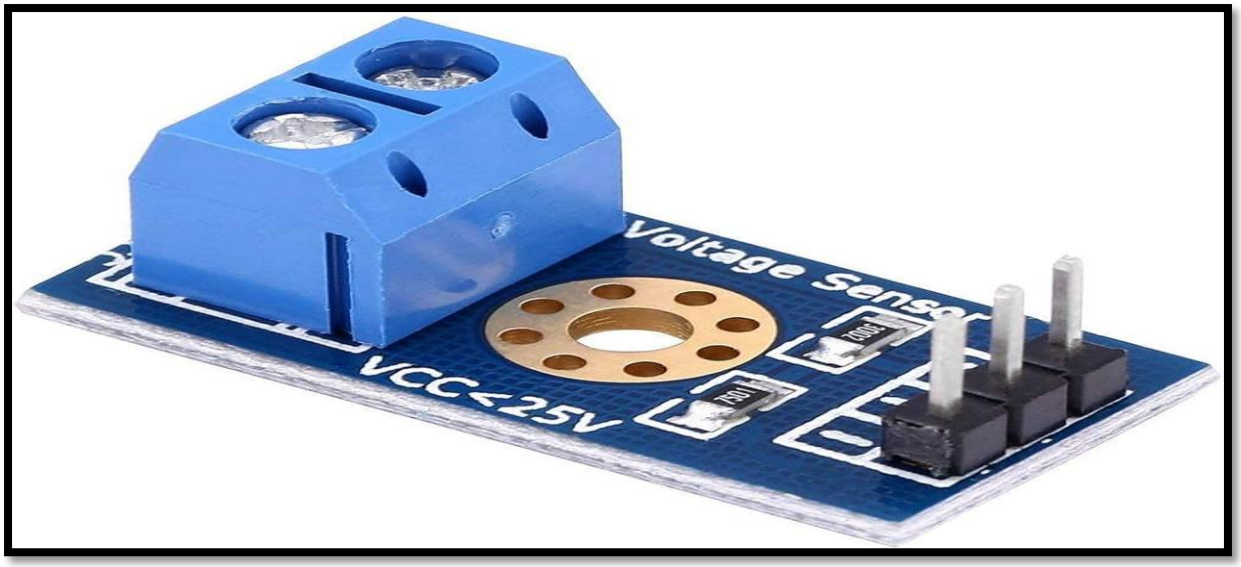


Figure 2.2: Voltage Sensor

The 25V voltage sensor is a highly efficient voltage sensor designed to measure and monitor DC voltage levels up to 25 volts. It is commonly utilized in applications such as battery monitoring, power supply testing, and voltage regulation systems. This sensor provides a convenient and accurate way to measure voltage levels and can be easily interfaced with microcontrollers like the Arduino UNO for real-time monitoring and data logging. The sensor's input voltage range spans from 0 to 25 volts, and it outputs a corresponding voltage of 0 to 5 volts DC, making it compatible with various microcontroller input levels. It boasts high measurement accuracy with a minimal error margin of ±1%, ensuring reliable and precise voltage readings. The 25V voltage sensor operates within a broad temperature range of -40°C to 85°C, maintaining consistent performance in diverse environmental conditions. Its compact dimensions (30mm x 20mm x 10mm) allow for easy integration into different projects. The sensor typically includes pins for power supply (VCC), ground (GND), positive voltage input (VIN+), and negative voltage input (VIN-).

These features make it an ideal choice for applications requiring accurate and efficient voltage level monitoring, such as in battery management systems, solar panel voltage measurement, and other critical voltage monitoring tasks.

2.3.4 Location capturing

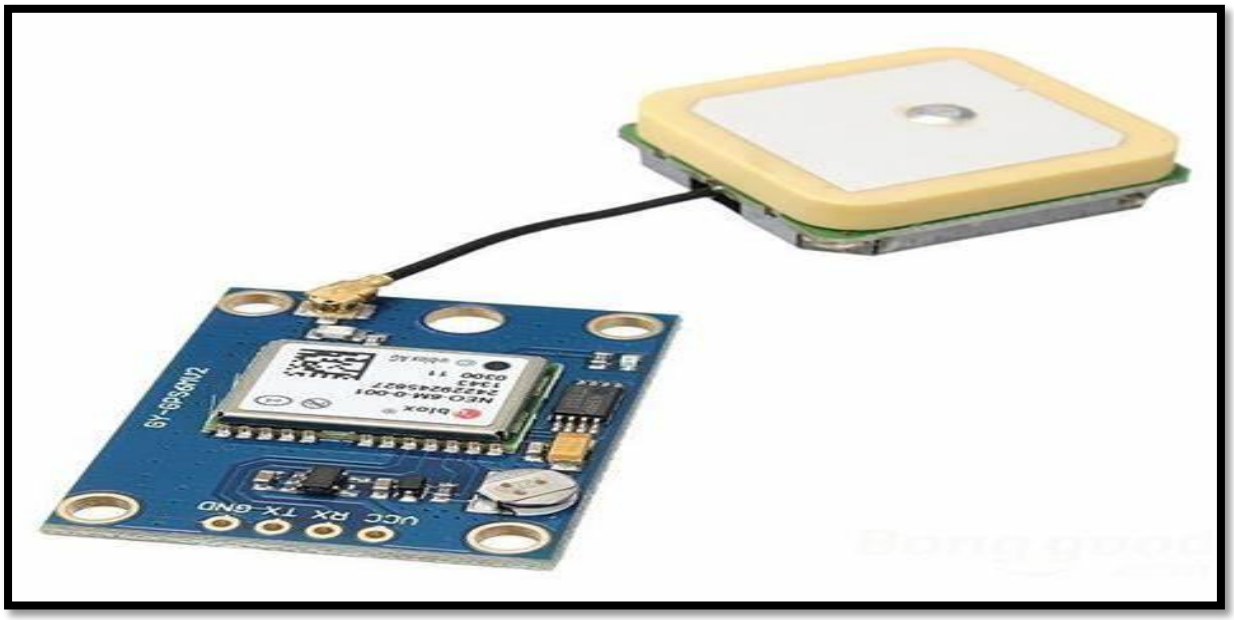


Figure 2.3: NEO6MV2 GPS module

The NEO-6M GPS module is a compact and high-performance satellite positioning receiver based on the u-blox NEO-6 series chipset. Designed to provide simple integration of GPS functionality into various electronic projects, the module can track up to 50 satellites

simultaneously, ensuring high sensitivity and improved accuracy. It features a 50-channel GPS L1 frequency receiver with a position accuracy of approximately 2.5 meters and an update rate of up to 5 Hz. The module boasts fast acquisition times, with cold starts taking 27 seconds, aided starts 3 seconds, and reacquisition 1 second. With a sensitivity of –161 dBm, it operates efficiently even in challenging conditions. It supports a wide operating temperature range from –40°C to 85°C and requires a supply voltage of 3.0V to 5.5V, consuming around 50mA at 5V. The NEO-6M GPS module includes essential pin configurations such as VCC (power supply), GND (ground), RX (UART receive), and TX (UART transmit). Its versatility and reliability make it suitable for applications like drones, vehicle tracking systems, personal navigation devices, and time synchronization, especially in battery-operated mobile devices where cost and space constraints are critical.

2.3.5 Liquid Crystal Display

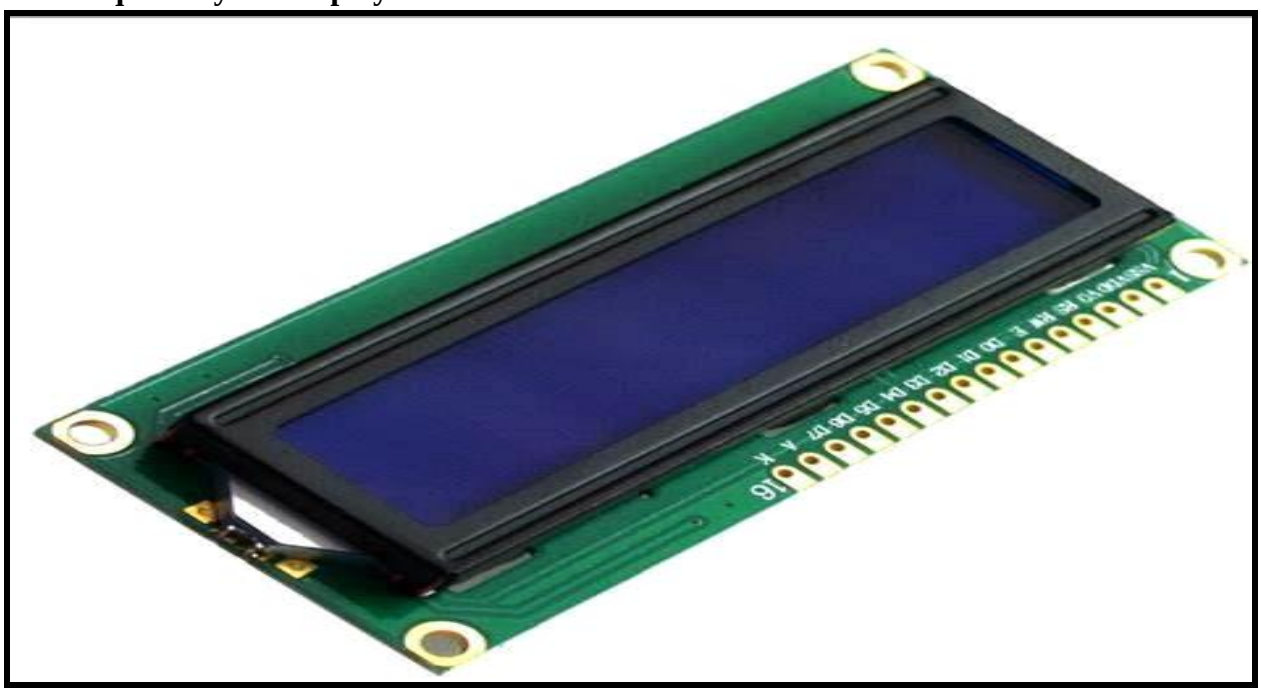


*Figure 2.4: 16*2 LCD display*

A micro controller in one Gi manipulates this parallel 16-pin interface in order to control the display. The interface is made up consisting of eight data pins, a register select pin, a read or write pin, and an enable pin. The register select pin selects a data register in the LCD memory, which holds the memory and an instruction register from which the micro controller for the LCD searches for instructions on the next step to take. The read and write pin essentially plays the function of selecting between two modes, which are read and write mode respectively. The

8 data pins exist in two states which are high and low. These determine the bits one is writing to a register or reading from a register. The screen turns on due to the use of power supply pins that use 5volts and LED backlight pins, which can turn it off. The display contrast pin, controls the display of the screen. The screen will be used in the project to display real time measured parameters and location.

2.3.6 Microcontroller Unit(ESP32)

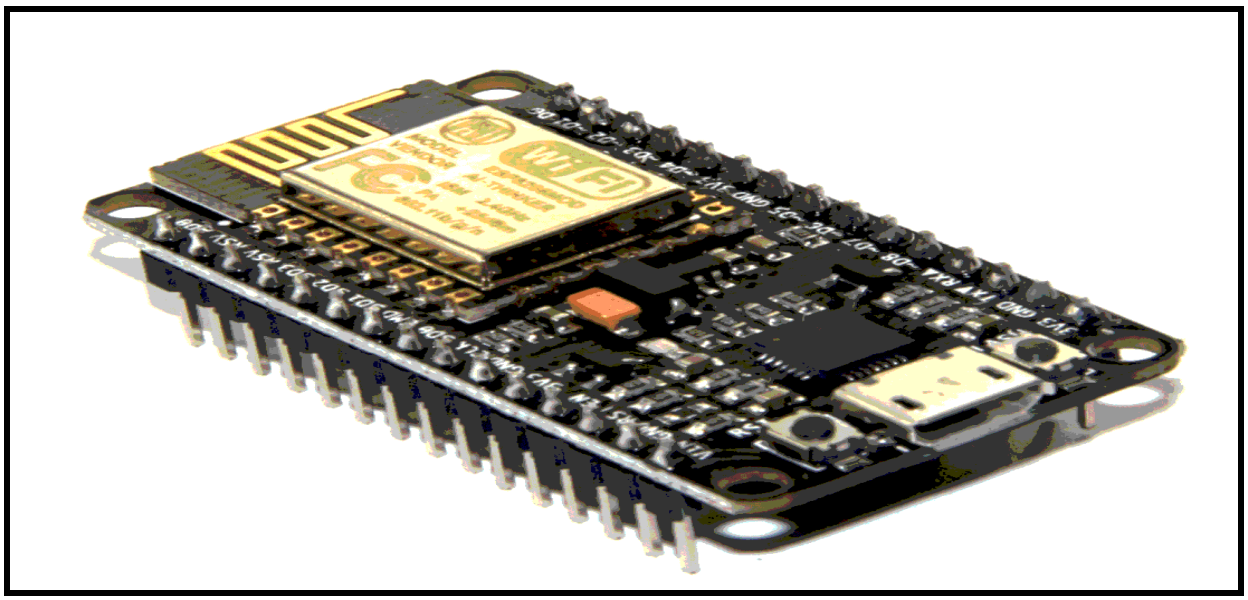


Figure 2.5: ESP32 microcontroller

The ESP32 microcontroller is a versatile and powerful device developed by Espressif Systems, designed to cater to a wide range of applications. It features a dual-core Xtensa LX6 microprocessor running at up to 240 MHz, providing robust performance for multitasking and real-time processing. The ESP32 comes with 512 KB of SRAM, 448 KB of ROM and 4 MB of Flash memory, making it suitable for complex applications.

One of the standout features of the ESP32 is its built-in Wi-Fi and Bluetooth module, which allows for seamless integration into IoT projects, home automation, and remote monitoring systems. Additionally, it supports a variety of peripheral interfaces, including SPI, I2C, UART, PWM, and ADC, enabling easy communication with sensors, displays, and other devices. The ESP32 also includes touch sensors, capacitive touch GPIOs, and a real-time clock, further enhancing its functionality.

Reasons for Using ESP32 in Our System:

- Dual-Core Processor: The dual-core architecture allows for efficient multitasking and improved performance, making it ideal for handling complex tasks.
- Built-in Connectivity: With integrated Wi-Fi and Bluetooth, the ESP32 simplifies the process of connecting to networks and other devices, reducing the need for additional modules.
- Rich Peripheral Set: The wide range of input/output options and peripheral interfaces makes it easy to integrate with various sensors and actuators.
- Low Power Consumption: The ESP32 is designed to be energy-efficient, with various power modes and sleep options, making it suitable for battery-powered devices.
- Arduino Compatibility: The ESP32 can be programmed using the Arduino IDE, providing access to a vast library of resources and community support.
- Cost-Effective: Despite its advanced features, the ESP32 remains affordable, making it accessible for both hobbyists and professionals.

2.4 Conclusion

Chapter 2 has provided an in-depth literature review covering the evolution and current state of battery management systems (BMS), real-time monitoring and data acquisition, GPS-based location tracking, and terminal security. By examining key studies and advancements in these areas, we have identified the limitations of existing technologies and highlighted the need for an integrated solution that addresses these challenges. The proposed system leverages advanced features such as real-time monitoring, GPS tracking, and robust security measures to enhance the efficiency, safety, and management of battery-powered systems. This comprehensive review not only establishes a solid foundation for our project but also underscores the significance of developing a cutting-edge BMS that meets the growing demand for reliable and innovative battery management solutions.

CHAPTER 3: SYSTEM DESIGN (METHODOLOGY)

3.1 INTRODUCTION

This chapter presents the design of an Integrated Battery Management System (BMS) that provides real-time monitoring, GPS-based location tracking, and advanced security features. The design begins with an overview of the system architecture, represented through a complete functional block diagram. This diagram is then broken down into smaller functional units to illustrate how each component contributes to the overall operation. The system is built around the ESP32 microcontroller, which coordinates sensor inputs and communication tasks. Key hardware components include the ACS712 current sensor, DS18B20 temperature sensor, 25V voltage sensor, and NEO-6M GPS module. These modules are grouped into core functional blocks such as power measurement, environmental sensing, GPS tracking and communication.

Each block is designed step-by-step. Voltage is measured through a voltage divider and ADC; current is sensed and calibrated using ACS712; temperature is read via the DS18B20's digital protocol; GPS data is processed from NMEA strings. The ESP32 is programmed to acquire, process, and transmit data wirelessly to a user interface. A mobile app, built using Ionic and Capacitor, provides real-time access to battery status and location. This structured design ensures each component integrates effectively, leading to a reliable BMS that addresses safety, efficiency, and user monitoring requirements.

3.2 OVERVIEW OF THE SYSTEM

The Battery Management System (BMS) is designed to monitor, manage and protect battery powered systems, ensuring their efficient and safe operation. It is crucial for applications such as electric vehicles, renewable energy storage and portable electronics, performing essential functions like real-time monitoring of battery parameters, balancing cell voltages, thermal management and implementing security measures. The system design includes the ESP32 microcontroller, which offers dual-core processing capabilities, built-in Wi-Fi and Bluetooth for seamless communication and various peripheral interfaces for easy sensor integration. The ACS712 current sensor accurately measures DC current with high sensitivity and electrical isolation, while the DS18B20 temperature sensor provides precise temperature monitoring with digital output and high accuracy. The 25V voltage sensor ensures reliable DC voltage measurement, and the NEO-6M GPS module provides real-time location tracking with high

precision. Key functions of the BMS include continuous real-time monitoring of current, voltage, and temperature to detect anomalies and trigger alerts for overcharging, deep discharging, and thermal runaway. The system ensures equal charge distribution among battery cells through cell balancing, extends battery life, and enhances performance. It also monitors and regulates battery temperature to prevent overheating, ensuring safe operation. Advanced security measures, include authentication mechanisms to protect against unauthorized access and data tampering through keypad access. The GPS module enhances asset management and theft prevention by providing accurate geolocation data. The user-friendly html webpage displays real-time battery parameters and location data, offering an intuitive interface for efficient system management. Overall, the BMS addresses the limitations of existing technologies by integrating advanced features, enhancing the efficiency, safety and management of battery-powered systems, making it an essential component for modern applications.

3.3 SYSTEM ARCHITECTURE: Block Diagram

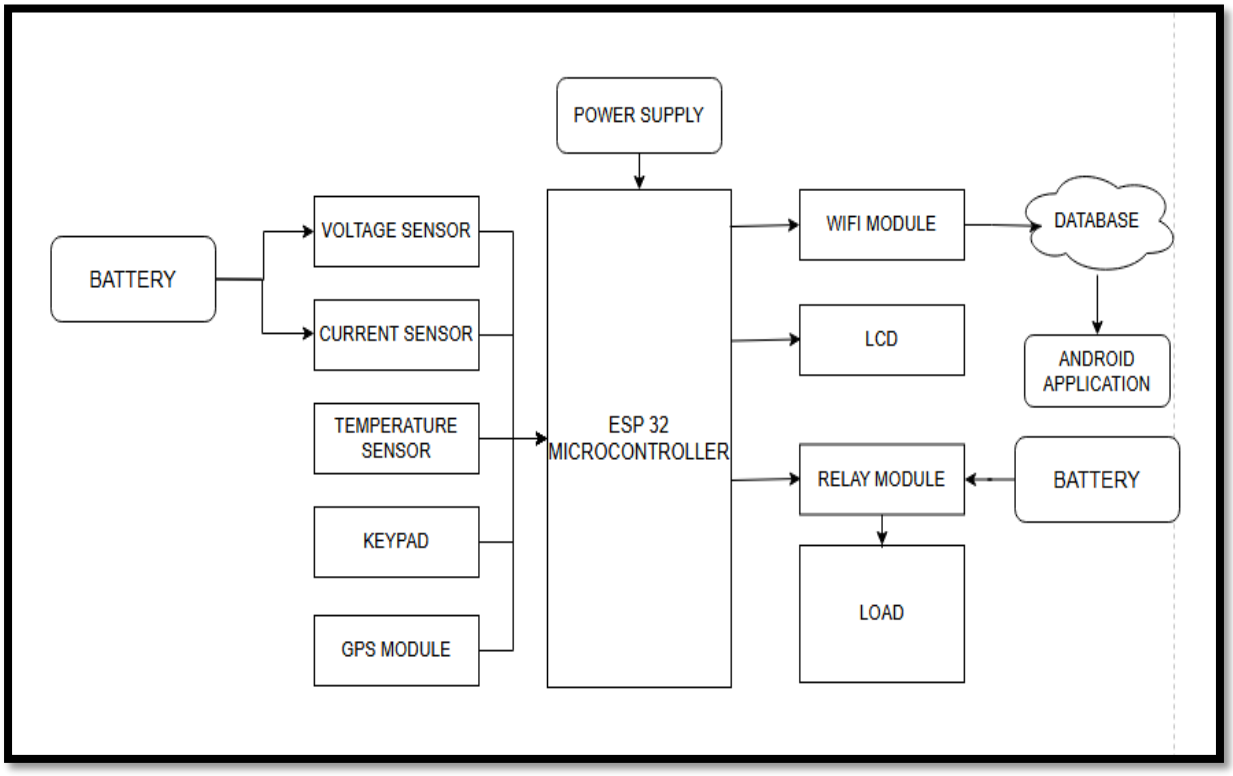


Figure 3.2: System Block Diagram

Figure 3.1 shows the complete system architecture of the access control system, the elements of this system is explained in the following page:

1. ESP32 Microcontroller: Acts as the central processing unit, handling data acquisition, processing sensor readings, executing control algorithms and managing communication via Wi-Fi and Bluetooth.
2. ACS712 Current Sensor: Measures the battery's charging and discharging current with high sensitivity and electrical isolation, helping to detect overcurrent conditions.
3. DS18B20 Temperature Sensor: Provides accurate digital temperature readings to monitor the battery's thermal condition and prevent overheating or thermal runaway.
4. 25V Voltage Sensor: Measures the battery voltage to ensure it stays within safe operating limits, preventing overcharging and deep discharging.
5. NEO-6M GPS Module: Enables real-time location tracking of the battery system, assisting in asset management and theft prevention.
6. Keypad Authentication System: Provides secure access control, ensuring only authorized users can interact with the system.
7. Battery Pack: The energy storage unit being monitored and managed for optimal performance and longevity.
8. Wi-Fi Module (Built into ESP32): Facilitates wireless communication for real-time data transmission to the web interface.
9. Android Application: Displays real-time battery parameters and location data, allowing users to monitor and manage the system remotely.

3.4 HARDWARE CIRCUIT DESIGN

The process of developing a fully functional circuit from individual component circuits involved carefully integrating each sensor and module into a single cohesive design. Initially, circuits were built separately for the ACS712 current sensor, DS18B20 temperature sensor, voltage sensor and NEO-6M GPS module, each tested independently with the ESP32 microcontroller to ensure proper operation. Once verified, the individual circuits were combined by connecting all sensor outputs to designated input pins on the ESP32, ensuring correct voltage levels, grounding and data communication protocols were maintained. Power management was also considered to supply stable voltage to all components. Pull-up resistors, capacitors and protective elements like diodes were added where necessary to improve reliability. This integration formed a complete, stable circuit that allowed simultaneous real time data acquisition from all sensors, enabling smooth system functionality for monitoring and control through the web and mobile applications.

3.4.1 Voltage and Current Sensor Configuration

3.4.1.1 Voltage Sensor Circuit

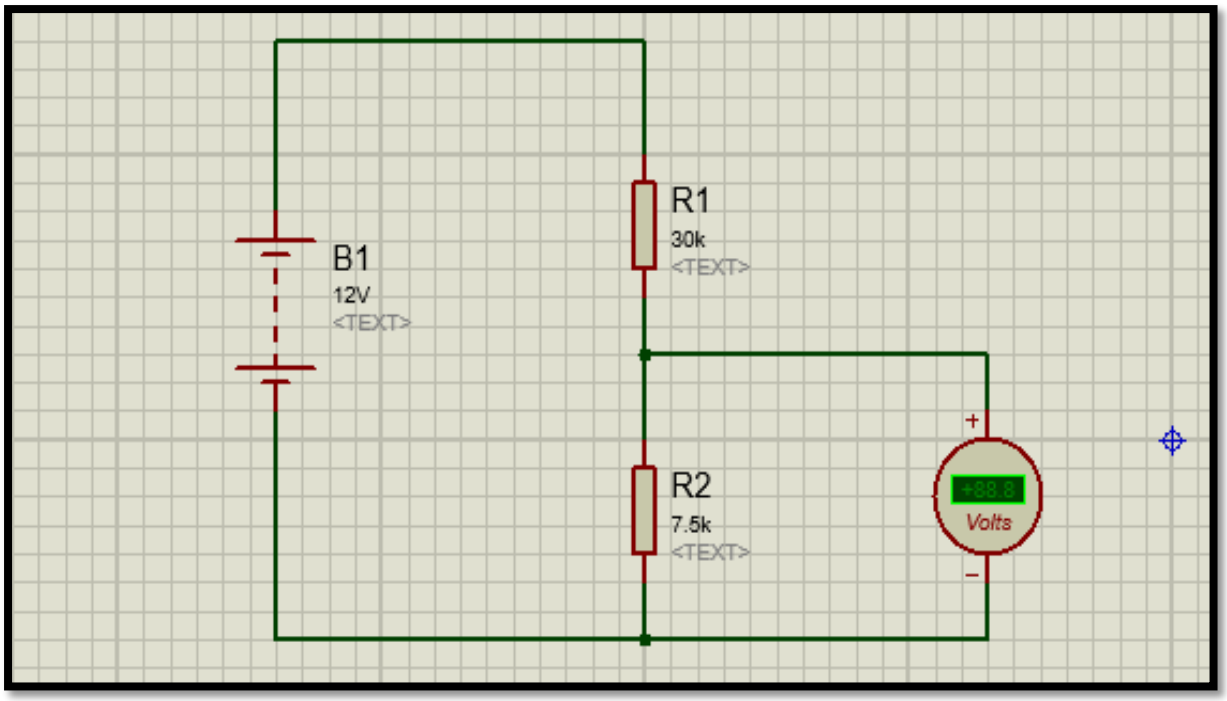


Figure 3.3: Voltage Sensor Circuit

Figure 3.2 above illustrates a simple voltage divider circuit used to scale down a higher input voltage to a lower level, which is particularly useful in microcontroller applications where analogue inputs are limited to 5V or 3.3V. In this circuit, a 12V DC supply (B1) is divided using two resistors in series: R1 = 30kΩ and R2 = 7.5kΩ. A voltmeter is connected across R2 to measure the output voltage.

Working Principle

A voltage divider works based on Ohm's Law and the principle of potential division in a series resistor network. When two resistors are connected in series across a voltage supply, the voltage drop across each resistor is proportional to its resistance.
The output voltage across R2 can be calculated using the voltage divider formula:

$$\mathbf{Vout} = \frac{R2}{R1+R2} \times \mathbf{Vin}$$

- Maximum Vin=12V
- Vout ≤ 3.3V(safe for ESP32 ADC)

With the following values we can have output voltage

- R1 = 30kΩ
- R2 = 7.5kΩ

Vout= $25\times\frac{7.5}{30+7.5}=12\times\frac{1}{5}=2.4V$

3.4.1.2 Voltage Sensor Configuration

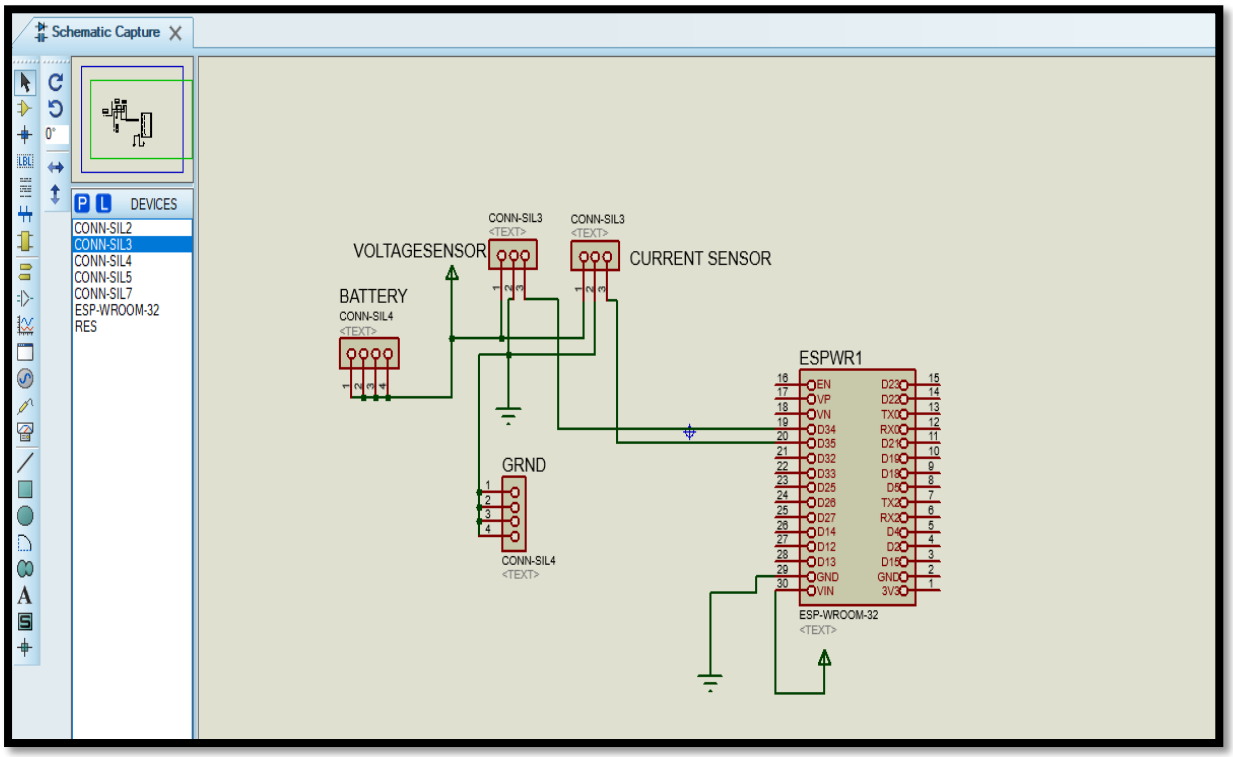


Figure 3.4: Voltage and Current Sensor Pin Configuration

Figure 3.3 shows how the voltage and current sensor are interfaced to the ESP 32 microcontroller.

The voltage sensor circuit is used to measure the battery voltage and ensure it remains within safe operating limits. It consists of a 25V voltage sensor module, which is essentially a voltage divider circuit designed to scale down higher voltages to a level that can be safely read by the ESP32's Analog-to-Digital Converter (ADC). In the circuit, the VCC pin of the voltage sensor is connected to the 3.3V supply from the ESP32, while the GND pin is connected to the system's common ground. The signal output (S pin) is connected to GPIO34 of the ESP32. The voltage sensor scales down the input voltage using a 5:1 resistor ratio, meaning the ESP32 reads only one-fifth of the actual voltage.

The measured ADC value is converted back to the real voltage using the equation:

Vactual=Vmeasured × 5V

This allows the system to accurately track battery voltage and detect overvoltage or deep discharge conditions. The readings are displayed on the LCD screen and transmitted to the android application via Wi-Fi for real-time monitoring.

3.4.1.3 Voltage Measurement Process Flow

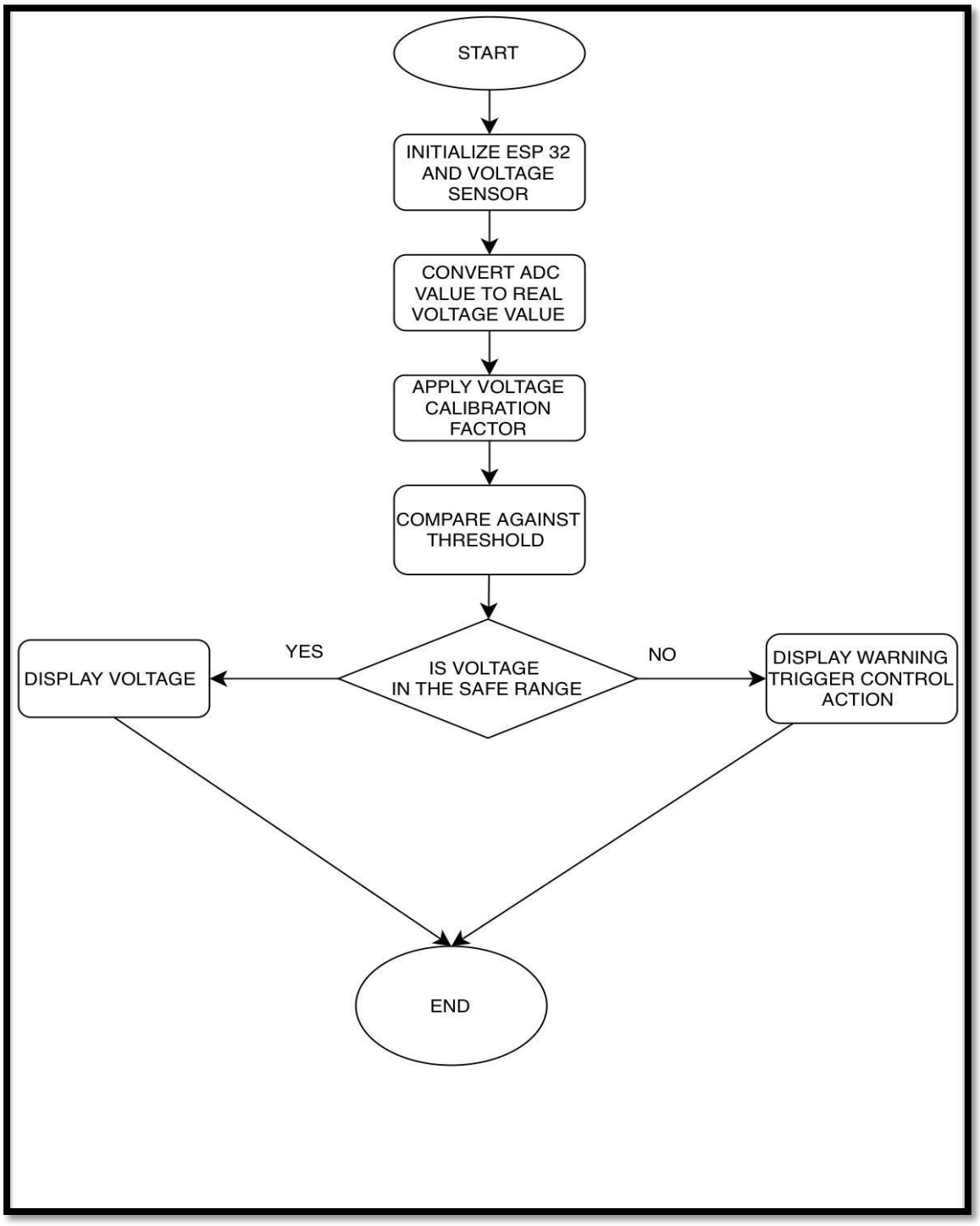


Figure 3.5: Voltage Sensor Flow

Figure 3.4 shows how the voltage sensor circuit acquires the voltage data from system initialisation.

This flowchart outlines the software logic used by your ESP32 to continuously monitor the battery voltage using an analog voltage sensor.

START

- This is the entry point of the program. The voltage monitoring system is about to begin

Initialize ESP32 and Voltage Sensor

- The ESP32 microcontroller boots up.
- The ADC (Analog-to-Digital Converter) pin is configured.
- The voltage sensor is powered and ready to read data.

. Convert ADC Value to Real Voltage Value

- The ESP32 reads a **raw ADC value** (between 0–4095 for 12-bit resolution).
- This value corresponds to the voltage level coming from the voltage divider.
- Formula:

$$V_{measured}=\left(\frac{ADC_{value}}{4095}\right)\times 3.3V$$

Apply Voltage Calibration Factor

- Since the voltage was scaled down by the voltage divider, you multiply the result by a calibration factor (e.g., 5 or 7.5 depending on your resistor ratio).
- Final voltage:

$$V_{actual} = V_{measured} \times \text{Calibratrion factor}$$

- This gives the true battery voltage.

Compare Against Threshold

- The calculated voltage is compared to safe operating limits (e.g., between 10V and 14.6V for a 12V battery).

Decision: Is Voltage in the Safe Range?

- Yes: Voltage is within safe range.
  Continue system operation and display the voltage on the LCD or app.
- No**:** Voltage is too high (overcharge) or too low (deep discharge)**.**
  Trigger warning alerts or protective actions (like cutting off charging).

If Safe → Display Voltage

- The voltage is shown on the LCD and possibly transmitted to the Android app.

If Unsafe → Display Warning & Trigger Action

- Show message like “Overvoltage!” or “Low Voltage!”
- Trigger a relay to disconnect the load or stop charging to protect the battery.

END

- The loop ends for one cycle.
- In reality, this loop restarts immediately for continuous monitoring

3.4.1.4 ESP32 Program for Voltage Measurement

```
void getVoltage() {
  adc_value = analogRead(36);
  adc_voltage = (adc_value * ref_voltage) / 1023.0;
  float bat_volt = adc_voltage / (R2 / (R1 + R2));
  Serial.println("Bat Volt:" + String(bat_volt));
  batPercent = map(bat_volt, 0.00, 45.0, 0.00, 100.0);
  current = map(batPercent, 0.00, 100.0, 0.0, 10.0);
  lcd.setCursor(0, 1);
  lcd.print("V:" + String(batPercent, 1) + "% C:" + String(current, 1) + "A");
  if (batPercent < 30) {
    digitalWrite(red, HIGH);
    digitalWrite(yellow, LOW);
    digitalWrite(green, LOW);
  }
  if (batPercent >= 30 && batPercent <= 75 ) {
    digitalWrite(red, LOW);
    digitalWrite(yellow, HIGH);
    digitalWrite(green, LOW);
  }
  if (batPercent > 75) {
    digitalWrite(red, LOW);
    digitalWrite(yellow, LOW);
    digitalWrite(green, HIGH);
```

3.4.2Current Sensor Configuration

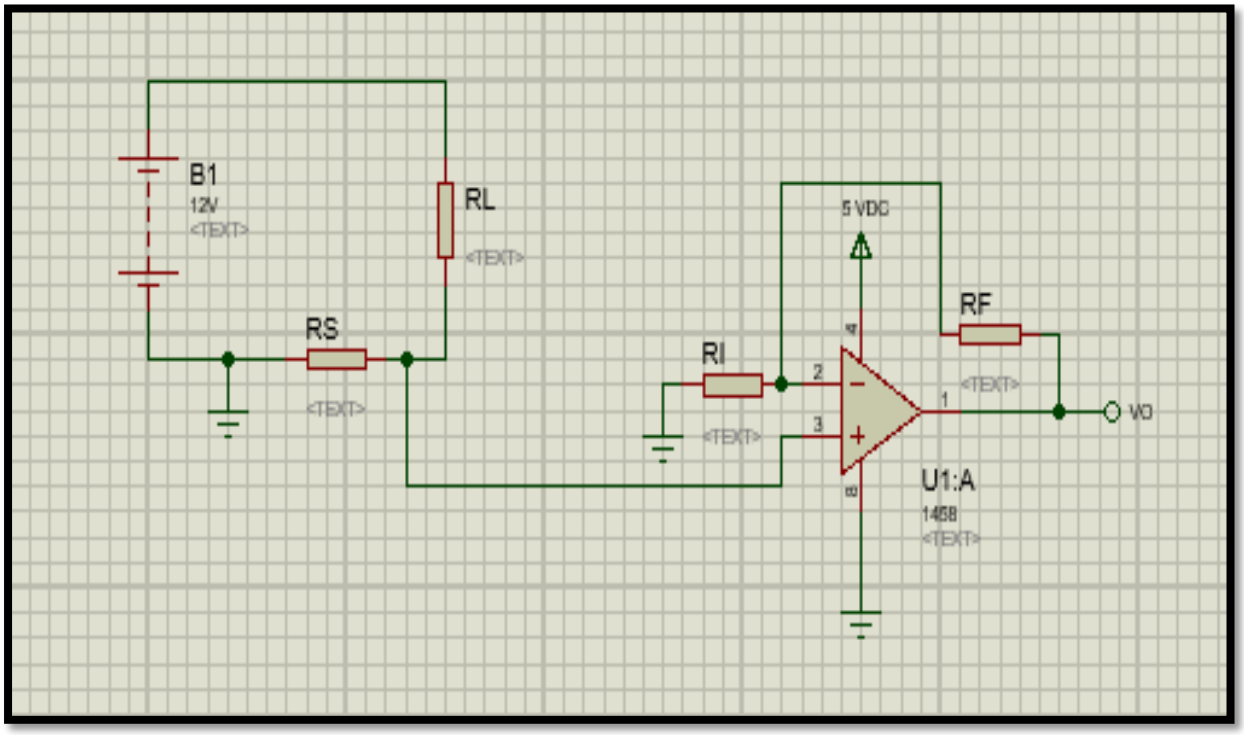


Figure 3.5: Current Sensor Measurement Circuit

Figure 3.6 illustrates a low-side current sensing circuit using a 0.1Ω shunt resistor (RS) and a differential amplifier built around an LM1458 op-amp (U1:A). The current flowing through the load (RL) creates a small voltage drop across RS, which is amplified by the op-amp to produce a readable analog output (Vo). The amplifier gain is set by the resistor ratio $\frac{RF}{R1}$=25 with R1=1kΩ and RF=25kΩ, allowing a 0.1V drop across RS (at 1A) to be amplified to 2.5V. This output can be fed into the ESP32's ADC for real-time current monitoring. The LM1458 op-amp is chosen for its dual-supply capability, input range including ground, and compatibility with 5V systems.

3.4.2.1 Current Measurement Process Flow

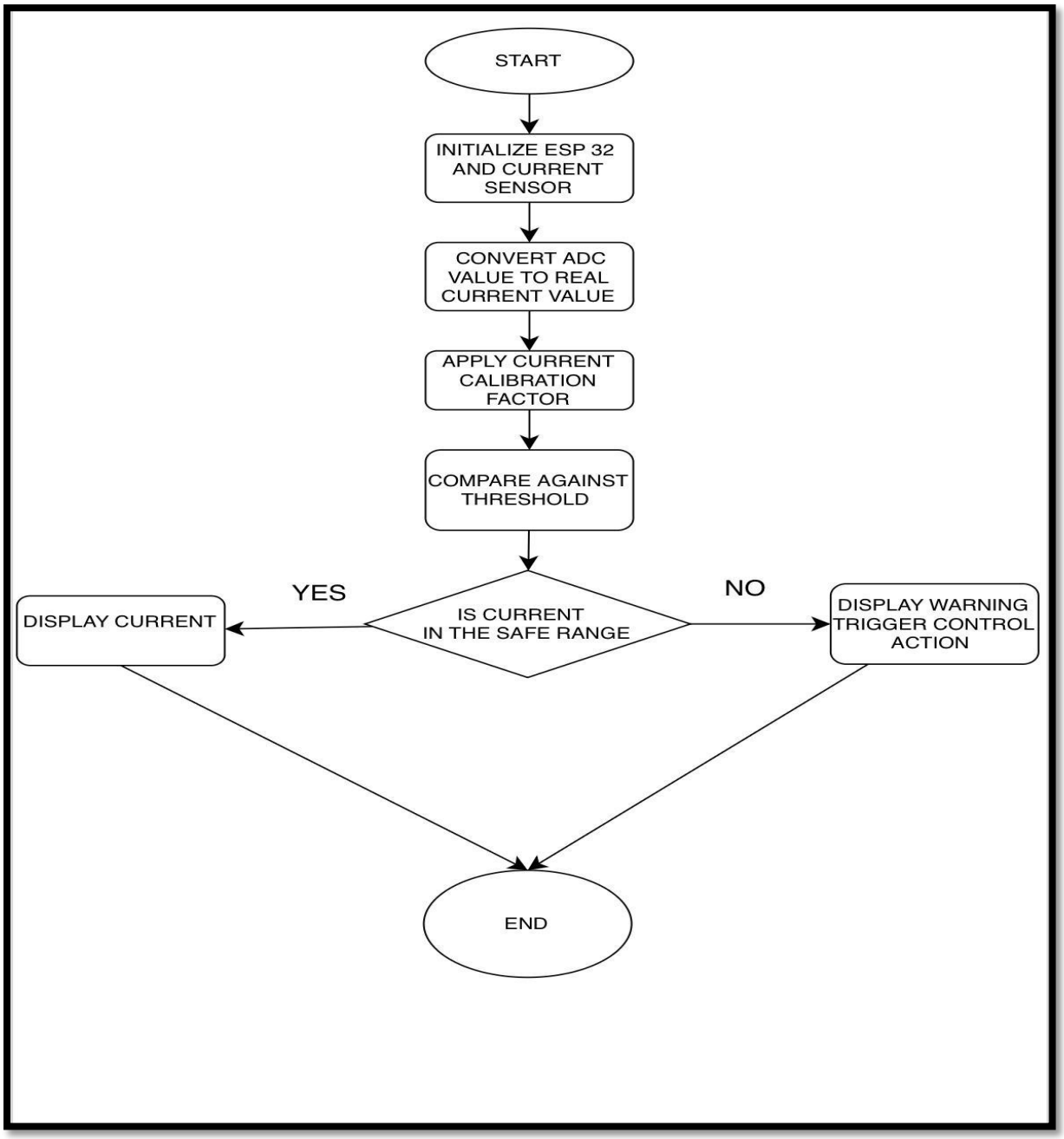


Figure 3.6: Current Measurement Flow

Figure 3.7 illustrates the logical flow for measuring and monitoring battery current using the ESP32 microcontroller and a current sensor such as the ACS712 or a shunt resistor and op-amp amplifier**.** This process allows the system to detect unsafe operating conditions like overcurrent or reverse current, and take appropriate action.

Start

The system begins execution when powered on or reset. Initialization is performed.

Initialize ESP32 and Current Sensor

The ESP32 configures the appropriate GPIO (ADC) pin for analog input. The current sensor (e.g. ACS712 or shunt +op-amp) begins outputting voltage proportional to current flow.

Convert ADC Value to Real Current Value

- The ESP32 reads a raw digital value (0–4095 for 12-bit ADC).
- This value is converted into voltage:

$$V_{measured} = \frac{ADC_{value}}{4095} \times 3.3V$$

Apply Current Calibration Factor

To convert the voltage into current, the following formula is used:

If using ACS712:

$I = \frac{V_{measured} - V_{offset}}{sensitivity}$

Where:

- Voffset=2.5V
- Sensitivity = 185 mV/A (for 5A sensor)

If using a shunt resistor + op-amp, then:

$$I = \frac{V_{out}}{Gain \times R_{shunt}}$$

Compare Against Threshold

The measured current is compared to predefined limits (e.g., 0A–5A). These are configured in software depending on battery specifications.

Is Current in the Safe Range?

- **Yes:** System is operating normally → current is displayed on LCD or mobile app.
- **No:** An abnormal condition exists (e.g., overcurrent, reverse current).

Display Warning / Trigger Control Action

When current exceeds safe range:

- Warning is shown on the LCD.
- Buzzer or LED is activated.
- Relay may cut off power to protect the battery or load.

End

This step marks the end of one measurement cycle. The system typically loops this process continuously to provide real-time current monitoring**.**

3.4.2.2 ESP32 Program for Current Measurement

The AC712 current sensor works by measuring the voltage value and convert it to the corresponding current value. The output voltage of the ACS712 is proportional to the current passing through it, with a sensitivity of 185mV/A for the 5A version.

Below is the code:

```
const int currentPin = 35;  // ADC input pin

const float sensitivity = 0.185; // ACS712-5A: 185 mV per Amp
```

```
const float adcVoltage = 3.3;
const int adcResolution = 4095;
const float offsetVoltage = 2.5; // Voltage at 0A current
void setup() {
  Serial.begin(115200);
  analogReadResolution(12);  // 12-bit ADC for ESP32
}
void loop() {
  int adcValue = analogRead(currentPin);
  float voltage = (adcValue * adcVoltage) / adcResolution;
  // Calculate current (in Amps)
  float current = (voltage - offsetVoltage) / sensitivity;
  // Display result
  Serial.print("Voltage: ");
  Serial.print(voltage, 3);
  Serial.print(" V, Current: ");
  Serial.print(current, 3);
  Serial.println(" A");
  // Optional: Trigger action if current is unsafe
  if (abs(current) > 5.0) {
    Serial.println("Overcurrent detected!");
    // Add relay cutoff or buzzer code here
  }
  delay(1000); // 1 second delay
```

}

3.4.3Temperature Sensor Configuration

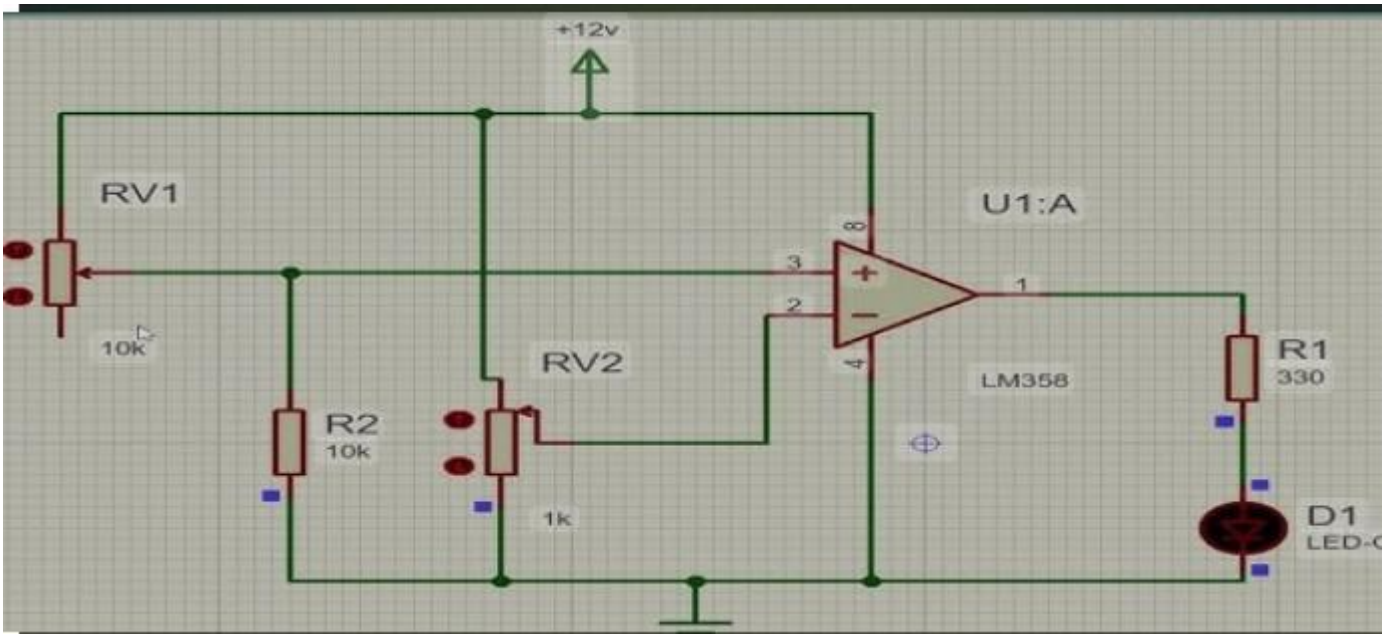


Figure 3.7: Temperature Sensor Measurement Circuit

Explanation and Analysis of Figure 3.7

1. Circuit Operation
Figure 3.7 represents a comparator circuit using the LM358 operational amplifier. This circuit is commonly used for threshold detection, where an output is switched ON or OFF depending on the voltage level at its inputs.

- RV1 and RV2 are variable resistors (potentiometers) used to adjust reference and input voltages:
  - RV1 and R2 form a voltage divider supplying a reference voltage to the non-inverting input (+) of the Op-Amp.
  - RV2 sets the input voltage connected to the inverting input (−).
- The LM358 compares these two voltages:
  - If the voltage at the non-inverting input (+) is greater than the voltage at the inverting input (−), the Op-Amp output goes high.
  - If not, the output remains low.
- The output from pin 1 drives an LED (D1) through a current-limiting resistor (R1, 330 Ω), indicating whether the input has exceeded the threshold.

2. Justification for Using LM358 Op-Amp
The LM358 is a dual-operational amplifier with the following advantages:

- It operates on single-supply voltage (3V to 32V), making it suitable for 12V systems.

- It can output voltages close to ground (zero) without needing negative supply.
- Low power consumption and widely available.
- Internally frequency compensated, ideal for comparator or amplifier applications.

3. Component Value Reasoning and Calculations

- R1 (330 Ω) limits the current through the LED. Assuming a 2V forward voltage for the LED and a 12V supply:

$$\mathrm{I} = \frac{12\mathrm{V} - 2\mathrm{V}}{330\Omega} = \frac{10\mathrm{V}}{330\Omega} = 30.3\mathrm{mA}$$

  This value is slightly high; for better LED protection, R1 can be increased to 470 Ω to limit current to ~21 mA.

- R2 and RV1 (10kΩ each) form a voltage divider:

$$\mathrm{V}_{\mathrm{ref}=\frac{\mathrm{RV1}}{\mathrm{RV1}+\mathrm{R1}}\times 12\mathrm{V}}$$

  When both are equal (10kΩ), the reference voltage is:

$$V_{ref=\frac{10k}{10k+10k}\times 12V=6V}$$

- RV2 (1kΩ) is adjustable, used to simulate varying sensor inputs (e.g., from a temperature sensor or voltage monitor).

4. Use Case in BMS Context
This comparator can be used to detect overvoltage, undervoltage, or temperature threshold. For example:

- If input voltage from a sensor (e.g., thermistor or voltage divider) exceeds a set limit, the LED indicates a warning or triggers a shutdown via GPIO.

3.4.3.1 Temperature Sensor Circuit

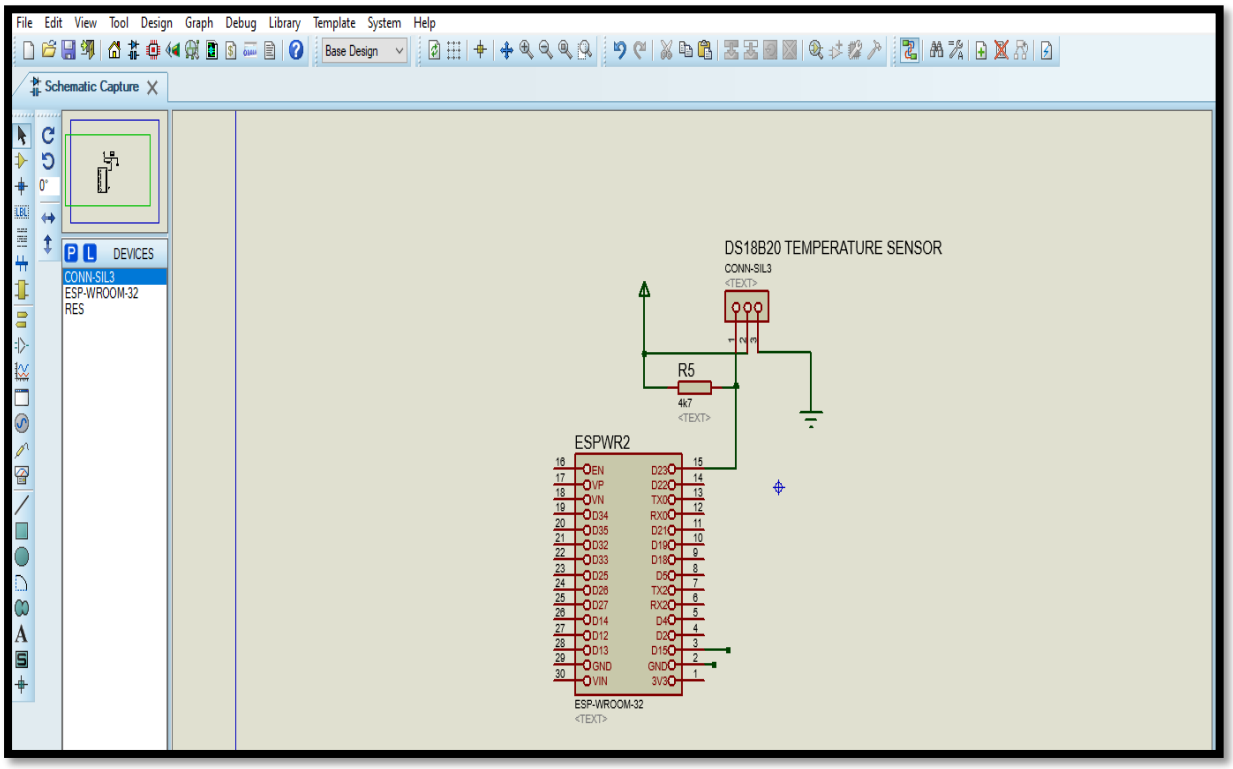


Figure 3.8: Temperature Sensor Circuit

Figure 3.9 shows the temperature sensor circuit in the Battery Management System (BMS) incorporating the DS18B20 digital temperature sensor to monitor battery temperature and prevent overheating.

The DS18B20 is a 1-wire sensor, meaning it requires only one data pin for communication with the ESP32 microcontroller, making it efficient and easy to integrate. In the circuit, the VCC pin of the DS18B20 is connected to the 3.3V supply from the ESP32, while the GND pin is connected to the common ground of the system. The data pin (DQ) is connected to GPIO23 of the ESP32, enabling digital temperature readings. A 4.7kΩ pull-up resistor is placed between the VCC and data pin to ensure stable communication and prevent floating signals.

The ESP32 reads the temperature data from the DS18B20 using the 1-wire communication protocol, which allows multiple sensors to be connected on the same data line if needed. The sensor provides high-accuracy temperature readings (±0.5°C) in a 9-bit to 12-bit resolution format, making it ideal for precise thermal monitoring. The acquired temperature data is used to trigger alerts if the battery temperature exceeds safe limits, preventing overheating and potential thermal runaway. The readings are displayed on the LCD screen and sent to the web

interface via Wi-Fi for remote monitoring. This integration ensures efficient temperature management, enhancing the battery's safety, lifespan and performance.

3.4.3.2Temperature Measurement Process Flow

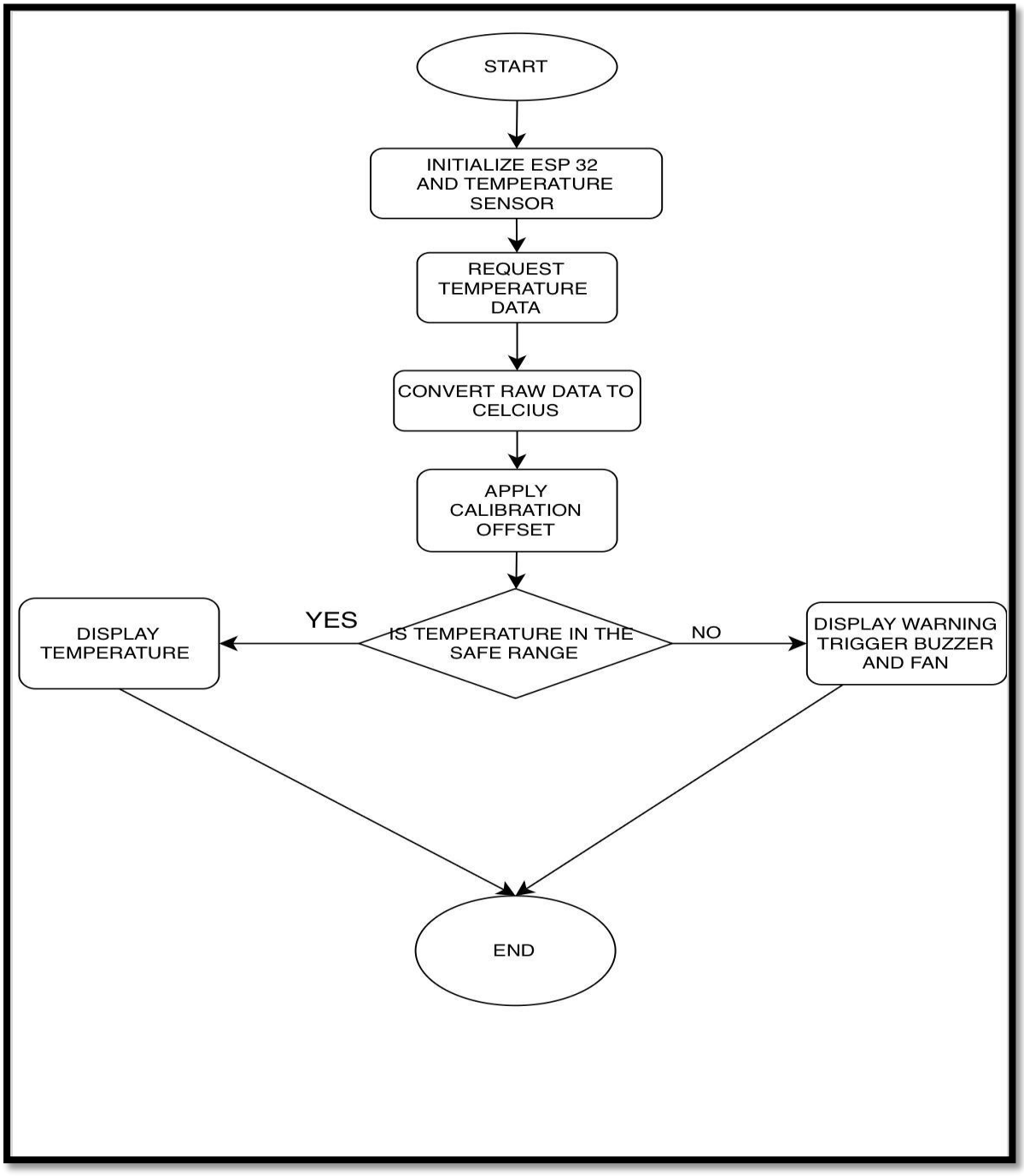


Figure 3.9: Temperature Measurement Process Flow

Figure 3.9 Explanation of Temperature Monitoring Flowchart

The flowchart illustrates the logical steps followed by the ESP32 microcontroller to monitor and respond to temperature changes in the system. The process is designed to ensure that temperature remains within a safe operating range. Below is a step-by-step explanation:

Start

The system begins operation when powered on or reset.

Initialize ESP32 and Temperature Sensor

The ESP32 microcontroller sets up its internal components and initializes communication with the temperature sensor (e.g., using I2C or analog input depending on the sensor type).

This step ensures the sensor is ready to send temperature data.

Request Temperature Data

The ESP32 sends a request to the sensor to read the current temperature.

The sensor responds with raw temperature data.

Convert Raw Data to Celsius

The raw data from the sensor is usually in a digital format or voltage that needs to be converted to a human-readable temperature in degrees Celsius.

This conversion depends on the sensor type (e.g., for LM35: Temp = voltage × 100).

Apply Calibration Offset

Some sensors may have slight inaccuracies. A known offset (based on testing) is applied to correct the value.

This helps ensure more accurate readings.

Is Temperature in the Safe Range?

The converted and calibrated temperature is compared against predefined safe limits (e.g., 20°C to 45°C).

If the temperature is within this range, the system proceeds to normal display. If not, it activates safety mechanisms.

If YES – Display Temperature

When the temperature is normal, it is displayed on an LCD screen for monitoring purposes.

If NO – Display Warning, Trigger Fan

If the temperature exceeds safe limits, the system activates:

- Cooling fan to reduce temperature.
- Warning message on the display to indicate an over-temperature condition.

End

The flow ends but typically loops back to continuously monitor the temperature at regular intervals.

3.4.3.3 ESP32 Program for Temperature Measurement

```
void batteryTemperature() {

 for (int i = 0; i < 4; i++) {
```

```
    sensor.requestTemperatures();
    Temp = sensor.getTempCByIndex(0);
    delay(200);
  }
  lcd.setCursor(0, 0);
  lcd.print("Temp:" + String(Temp));
  lcd.write(2);
  lcd.print("C");

  if (Temp > 27) {
    digitalWrite(fan, LOW);
  } else {
    digitalWrite(fan, HIGH);
  }

}
```

3.4.4 GPS Module Configuration

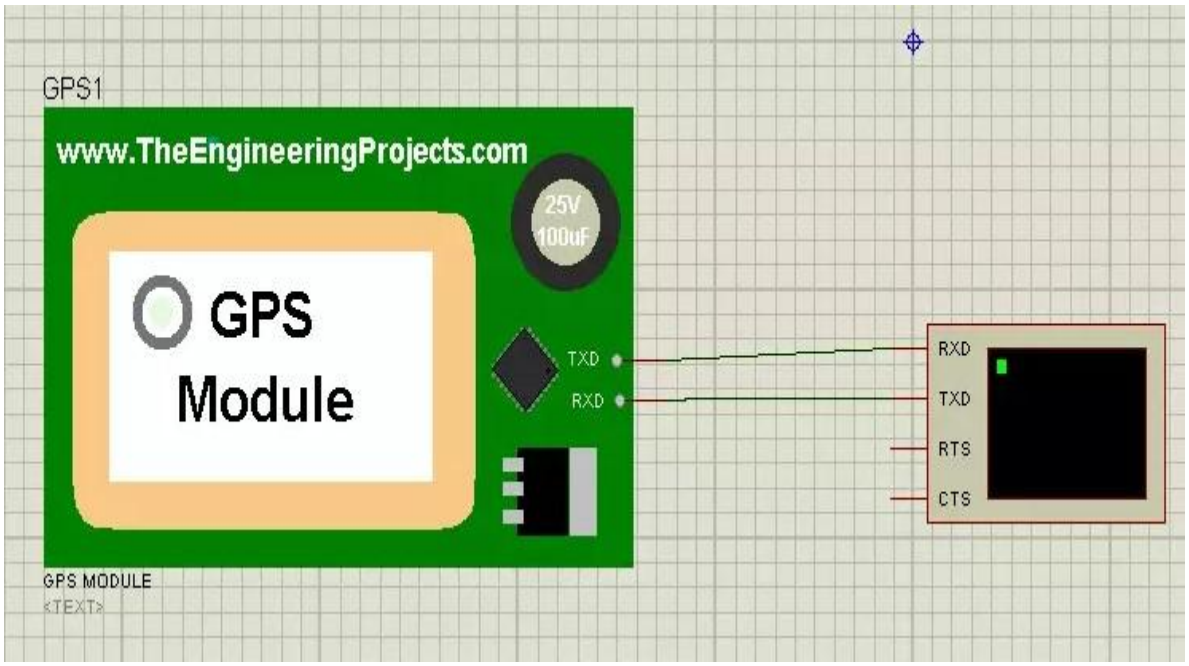


Figure 3.10: GPS module configuration

Figure 3.10 shows the GPS module (Neo-6M) connected to a virtual terminal via UART. The TX pin of the module is connected to the RX of the serial terminal to allow the GPS to send location data. The Neo-6M was chosen due to its compact size, reliable signal acquisition, low cost, and compatibility with standard microcontrollers through UART communication.

3.4.4.1 GPS Module Configuration

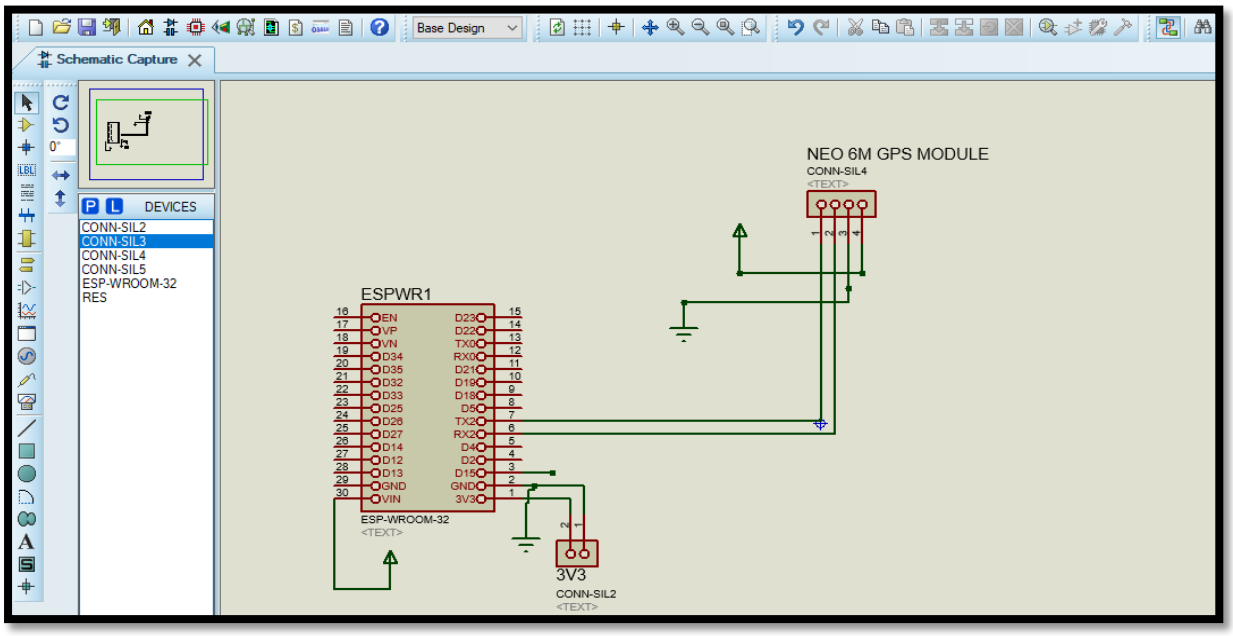


Figure 3.6: GPS module configuration

Figure 3.12 shows the NEO-6M GPS module configuration. This is used for real-time battery location tracking, helping with asset management and theft prevention.

The module communicates with the ESP32 using the UART protocol. In the circuit, the VCC pin of the GPS module is connected to the 3.3V supply from the ESP32, while the GND pin is connected to the system's common ground. The TX (Transmit) pin of the GPS is connected to GPIO16 (RX) of the ESP32, and the RX (Receive) pin is connected to GPIO17 (TX), allowing bidirectional data communication. The GPS module continuously sends latitude and longitude to the ESP32, which processes this information and displays the real-time location on the LCD and web interface.

3.4.4.2 ESP 32 Program to read GPS data

```
void loop() {
  if (millis() - lastGpsTime >= gpsTimeout) {
    lastGpsTime = millis();
    boolean newData = false;
    for (unsigned long start = millis(); millis() - start < 1000;) {
```

```
    while (mygps.available()) {
      if (gps.encode(mygps.read())) {
        newData = true;
      }
    }
  }
  if (newData == true) {
    newData = false;
    if (gps.location.isValid() == 1) {
      // GPS data is valid
      latitude = gps.location.lat();
longitude = gps.location.lng();
    } else {
      // If GPS data is invalid, use the default location
      latitude = defaultLatitude;
      longitude = defaultLongitude;
    }
    lat_str = String(latitude, 12);   // latitude location is stored in a string
    lng_str = String(longitude, 12);  // longitude location is stored in a string
    speed = gps.speed.kmph();
    speed_str = String(speed, 2);
    locString = "https://www.google.com/maps/search/?api=1&query=" + lat_str + "," +
lng_str;
    Serial.println(locString);
    digitalWrite(c1, LOW);
```

```
digitalWrite(c2, HIGH);
digitalWrite(c3, HIGH);
if (digitalRead(r4) == LOW) {
  tone(buzzer, 100, 100);
  lcd.clear();
  checkPassword();
}
```

The code shows an Arduino IDE designed to interface with a GPS module using the TinyGPS++ library for extracting and displaying location data. The code starts by including the TinyGPS++.h library, which simplifies parsing of GPS data received over serial communication. A TinyGPSPlus object named gps is created, and a HardwareSerial port (gpsSerial) is initialized on UART1 of the microcontroller using GPIO pins 16 (RX) and 17 (TX), commonly used on ESP32 boards. In the setup() function, the main serial port is started at 9600 baud for output to the serial monitor, and the GPS serial port is also initiated at 9600 baud to match the GPS module's default communication speed. A confirmation message "GPS Module Initialised" is printed to the serial monitor to indicate successful startup. In the loop() function, the code continuously checks if GPS data is available and reads it using the gps.encode() method to process the incoming NMEA sentences. If a valid GPS fix is detected using gps.location.isUpdated(), the latitude and longitude coordinates are extracted and printed to the serial monitor with six decimal places of precision. This program is essential for real-time location tracking applications, such as asset monitoring or navigation systems, and plays a key role in systems like IoT-based battery management solutions that require geographical context for remote deployments.

3.4.5 Keypad Module Configuration(Pull down circuit for key press)

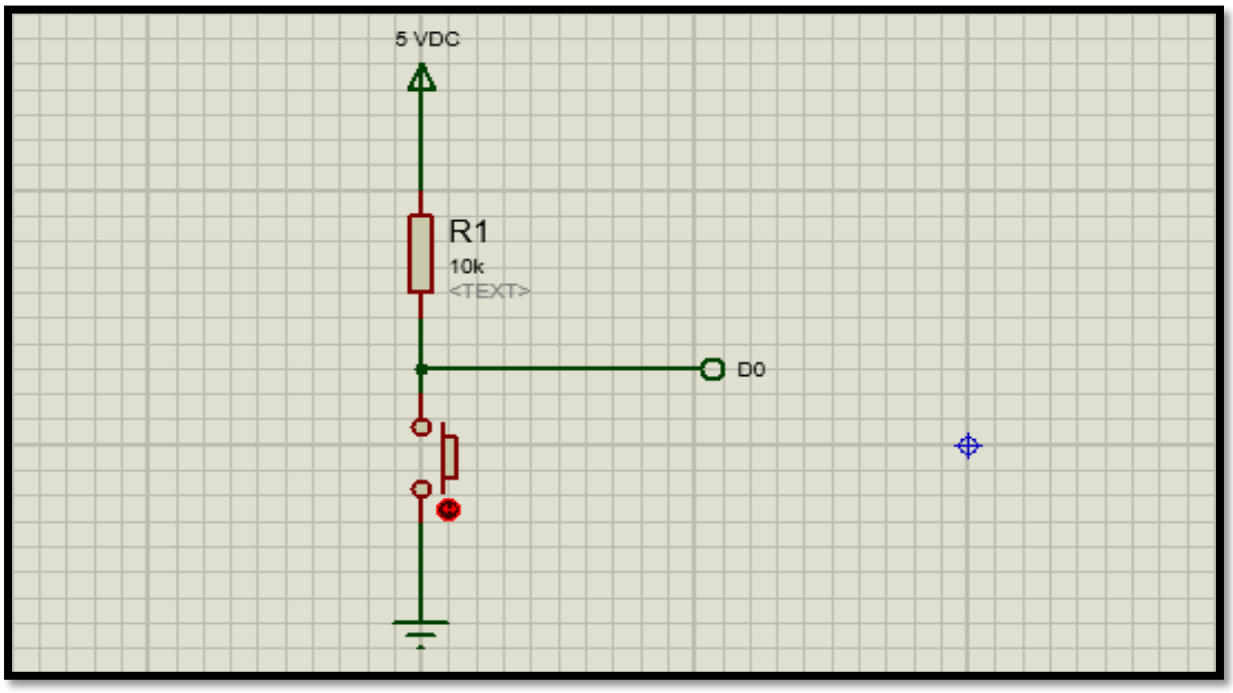


Figure 3.7: Pull down resistor circuit

Figure 3.14 Explanation of Pull-Down Resistor Circuit

The circuit shown in Figure X.X demonstrates the use of a pull-down resistor connected to a digital input pin (D0). The 10kΩ resistor (R1) is connected between the input pin and ground, while a push-button connects the pin to +5VDC when pressed.

Purpose of the Pull-Down Resistor

The pull-down resistor ensures that the digital input (D0) has a defined logic level (LOW = 0V) when the button is not pressed. Without this resistor, the pin could float and pick up noise, leading to unpredictable or false triggering.

How It Works

- Button Not Pressed: D0 is connected to ground via the resistor → logic LOW (0).
- Button Pressed: +5V is applied directly to D0 → logic HIGH (1)

Why 10kΩ?

A **10kΩ** resistor is commonly used because it provides enough resistance to avoid excessive current draw when the button is pressed, while still strong enough to pull the pin LOW when the button is released.

3.4.5.1 Keypad Configuration

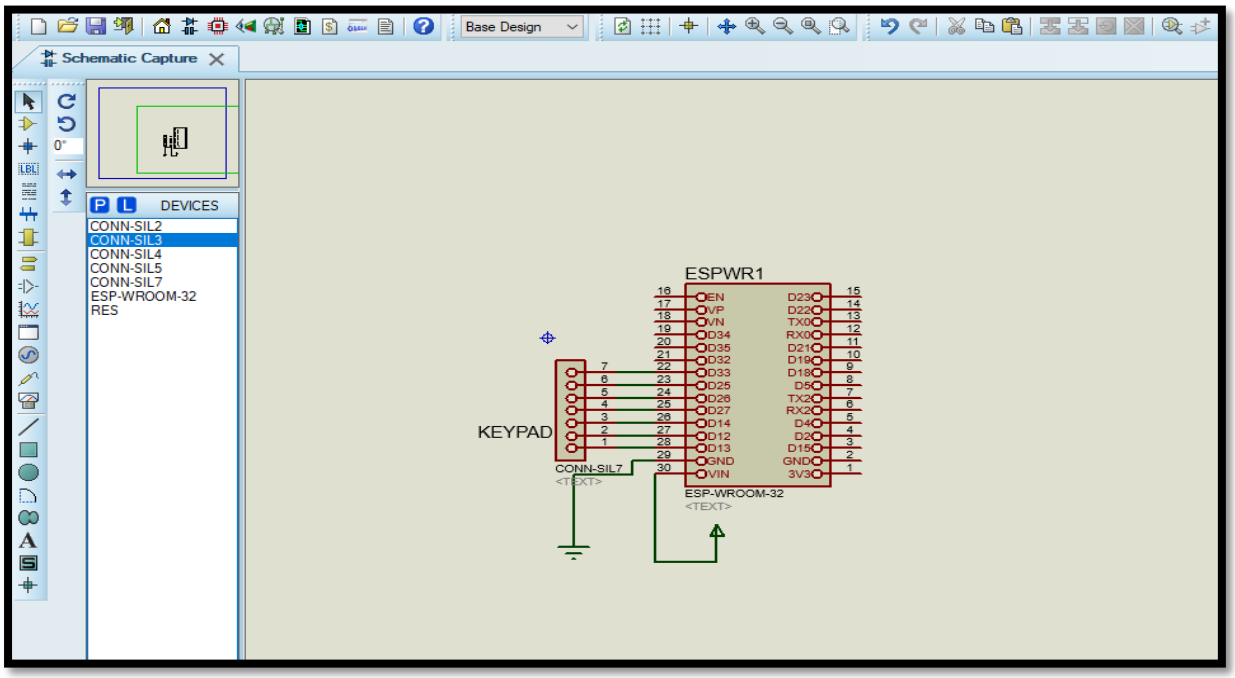


Figure 3.8: Keypad Module Configuration

Figure 3.15 shows the interfacing of a 4x4 matrix keypad with the ESP32 microcontroller **(**ESP-WROOM-32) in Proteus. The keypad allows the user to input numeric or control commands into the system, such as authentication for secure access.

How the Keypad Works

- The 4x4 keypad consists of 4 rows and 4 columns, forming 16 button positions.
- Each key press connects a specific row and column, forming a switch matrix.
- By scanning the rows and columns, the ESP32 detects which key is pressed.

Wiring Description

- The 8 pins of the keypad (4 rows + 4 columns) are connected to GPIO pins of the ESP32.
- In the schematic, the keypad is connected to a CONN-SIL7 connector, which then links to various digital I/O pins of the ESP32 (D32 to D27 and D14, etc.).
- A common ground connection ensures proper reference for logic levels.

Purpose of This Configuration

- This setup enables the microcontroller to scan key presses using software (with the Keypad library in Arduino or equivalent).
- It is essential for features like:
    - User input
    - Password protection
    - Menu navigation

3.4.5.2 Password Verification Flowchart

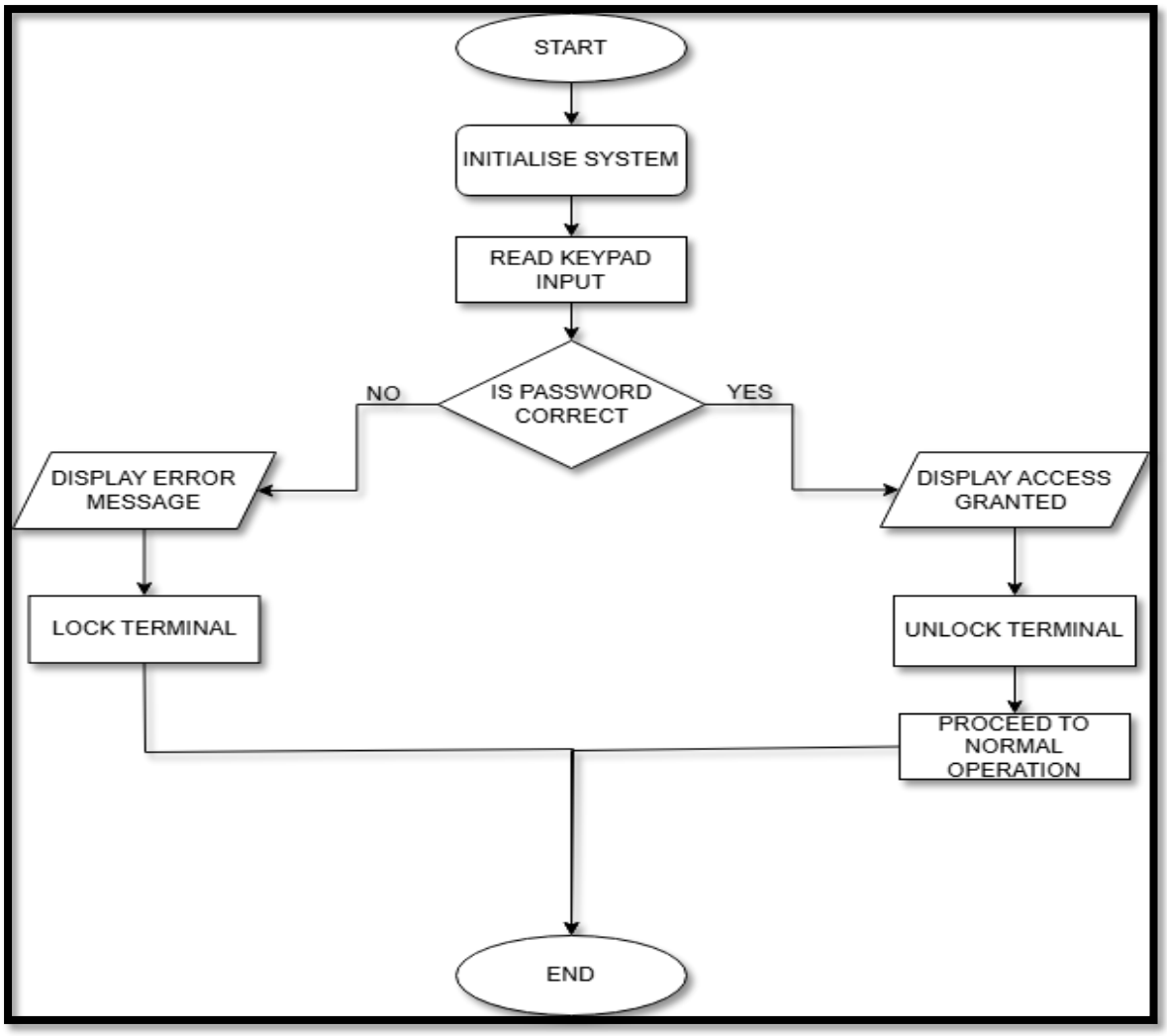


Figure 3.9: Password Verification Flow

Figure 3.16 entails the password verification system for unlocking the battery terminals.

The process begins with the user entering a password through a 4x4 keypad. Each digit is entered sequentially and displayed as an asterisk (*) for security. Once the password input is complete, the system retrieves the stored password from the ESP32 microcontroller for comparison. If the entered password matches the stored value, the system grants access by activating a relay module which in turn unlocks the battery terminals, allowing power to flow. A confirmation message, “Access Granted,” is displayed on the LCD screen and a green LED indicator lights up to signal successful authentication. However, if the password is incorrect, the system denies access, displaying “Incorrect Password” on the LCD while keeping the battery terminals locked. A red LED indicator turns on to signify a failed attempt. This mechanism ensures that only authorized users can access the battery system, enhancing security and preventing unauthorized tampering.

3.4.5.2 ESP 32 Program for Keypad Initialization

```
#include <Keypad.h>

const byte ROWS = 4; // Four rows

const byte COLS = 4; // Four columns

char keys[ROWS][COLS] = {

  {'1','2','3','A'},

  {'4','5','6','B'},

  {'7','8','9','C'},

  {'*','0','#','D'}

};

byte rowPins[ROWS] = {12, 13, 14, 27}; // Connect to the row pinouts of the keypad

byte colPins[COLS] = {32, 33, 25, 26}; // Connect to the column pinouts of the keypad

Keypad keypad = Keypad(makeKeymap(keys), rowPins, colPins, ROWS, COLS);

void setup() {

  Serial.begin(115200);

}

void loop() {

  char key = keypad.getKey();

  if (key) {

    Serial.println(key);

  }

}
```

3.4.6 Liquid Crystal Display Configuration

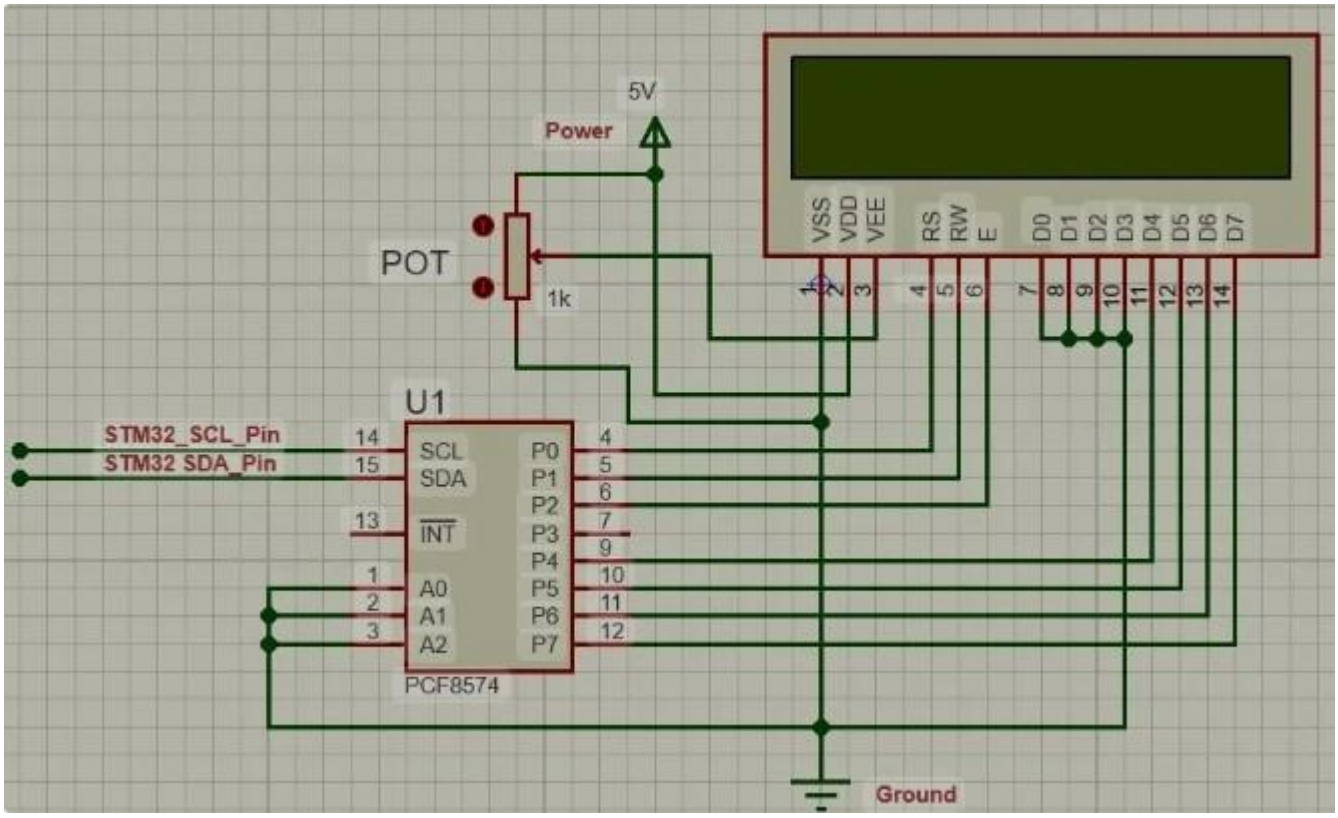


Figure 3.10: LCD Configuration schematic

The figure 3.18 shows the interfacing of a 16x2 LCD display with an STM32 microcontroller through the PCF8574 I/O expander using the I2C protocol. This configuration reduces the number of GPIO pins required by the LCD from 6–8 to just 2 (SCL and SDA). The PCF8574 translates I2C signals from the STM32 into parallel signals for the LCD. A 1kΩ potentiometer is used to adjust the contrast of the LCD through the VEE pin. This setup simplifies wiring and allows for easy expansion, making it ideal for embedded systems with limited I/O resources

3.4.6.1 Liquid Crystal Display Configuration

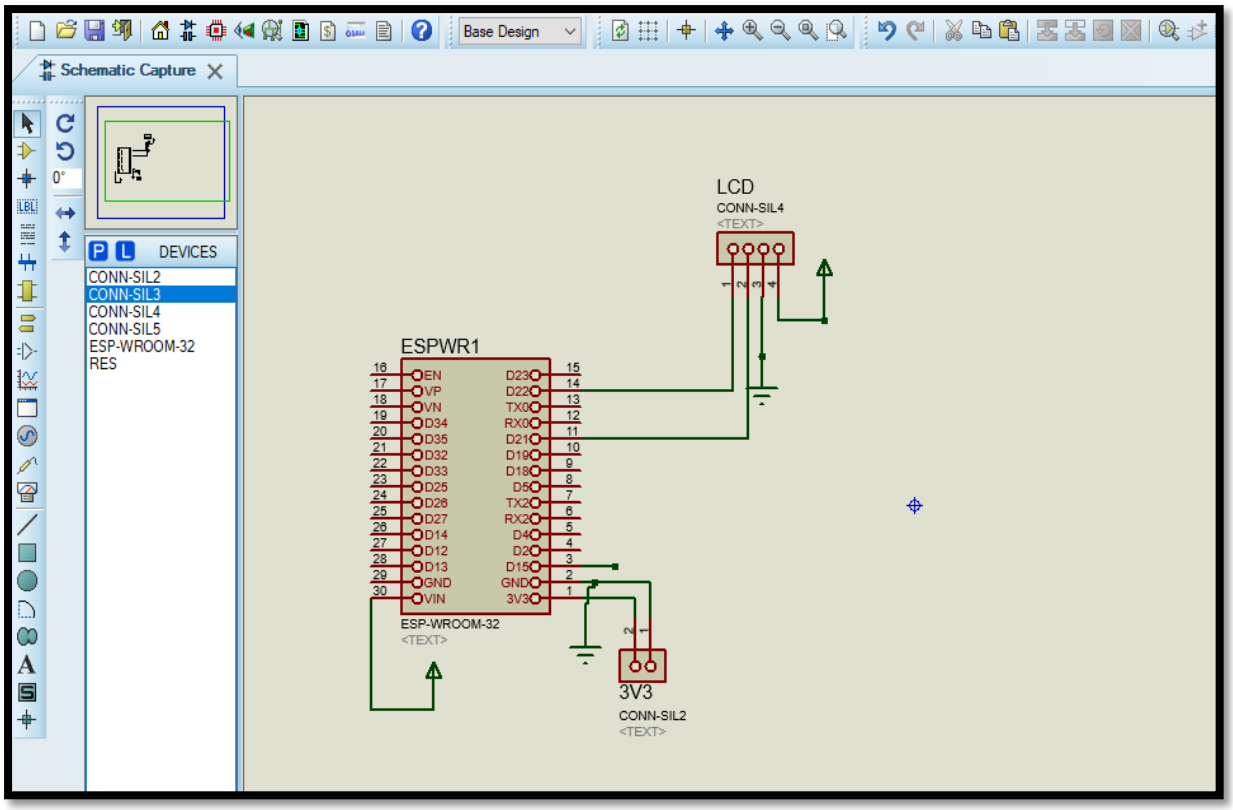


Figure 3.11: LCD Configuration via 12C

Figure 3.18 shows how the 16x2 LCD display is used to show real-time battery parameters such as voltage, current, temperature and system status.

This I2C-based connection allows efficient data transmission with minimal wiring, making it easy to integrate additional sensors while preserving GPIO availability. The LCD continuously updates the battery status, sensor readings, and warning messages, ensuring real-time system monitoring and quick user feedback.

3.4.7 Buzzer and LED Configuration

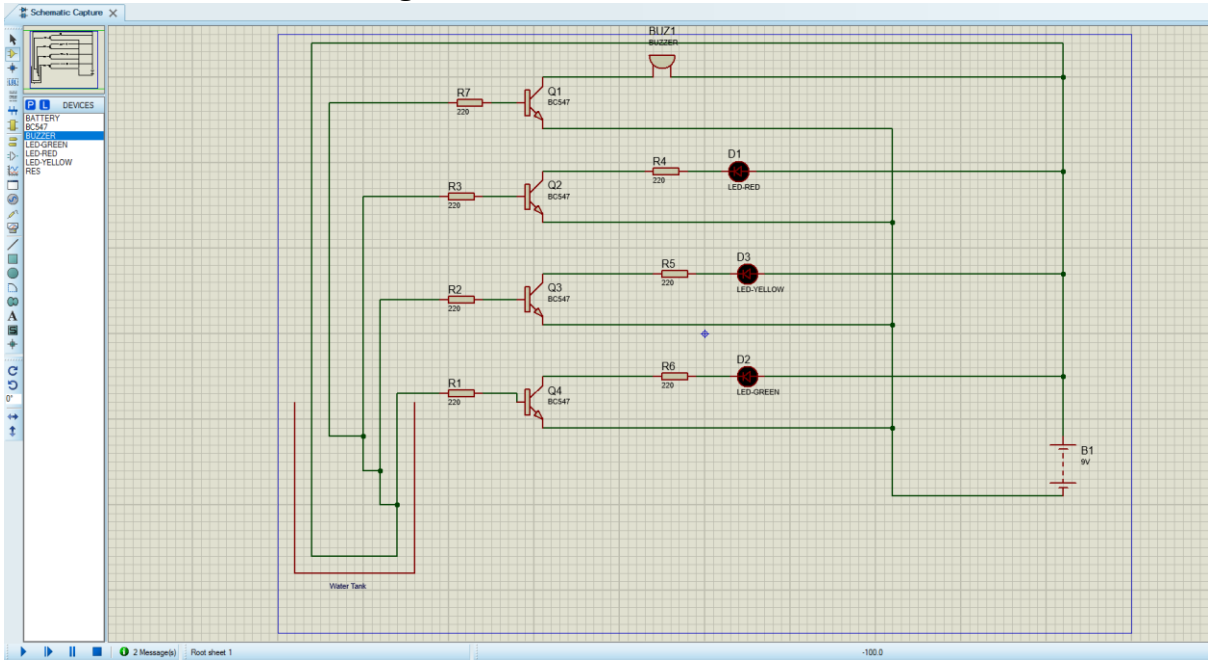


Figure 3.19: Buzzer and LED Configuration

Figure 3.19 presents a voltage level monitoring circuit using LEDs and a buzzer to provide visual and audio indication of a battery's charge status. This design uses BC547 NPN transistors (Q1–Q4) to switch the LEDs and buzzer ON based on the voltage level sensed at the base of each transistor.

The configuration works as follows:

- The battery voltage is divided using a resistive network (not shown in this simplified diagram) and connected to the base of each transistor (Q1–Q4) through resistors R1 to R4.
- As the battery voltage drops, different transistors become active depending on their base threshold voltage.

Operation:

- Green LED (D2) - Full Charge:
  When the battery voltage is high (e.g., >7.5V), transistor Q4 is turned ON. This allows current to flow through R6 and light up the Green LED, indicating that the battery is fully charged.
- Yellow/Orange LED (D3) - Moderate Level:
  As voltage drops to a moderate level (e.g., 6.5V–7.5V), transistor Q3 is triggered. The Yellow LED (D3) turns ON via R5, signaling that the battery is at a medium charge level.
- Red LED (D1) - Low Battery:
  When voltage drops further (e.g., below 6.5V), transistor Q2 switches ON, allowing the Red LED (D1) to illuminate via R4, indicating low battery condition.
- Buzzer (BUZ1) - Critical Level:
  If the voltage drops to a critical level (e.g., below 6V), transistor Q1 is activated,

triggering the buzzer through R7. The buzzer serves as an audio alert that immediate charging is required to avoid shutdown or damage to the battery.

Each transistor acts as a voltage threshold switch, turning ON only when the input voltage at its base exceeds approximately 0.7V (the base-emitter threshold of NPN transistors). Current-limiting resistors (R4–R7) are used to protect LEDs and the buzzer from excessive current.

The system is powered by a 9V battery (B1) and can be integrated into a Battery Management System (BMS) as a simple yet effective status indicator module.

3.4.7.1 Buzzer and LED Configuration

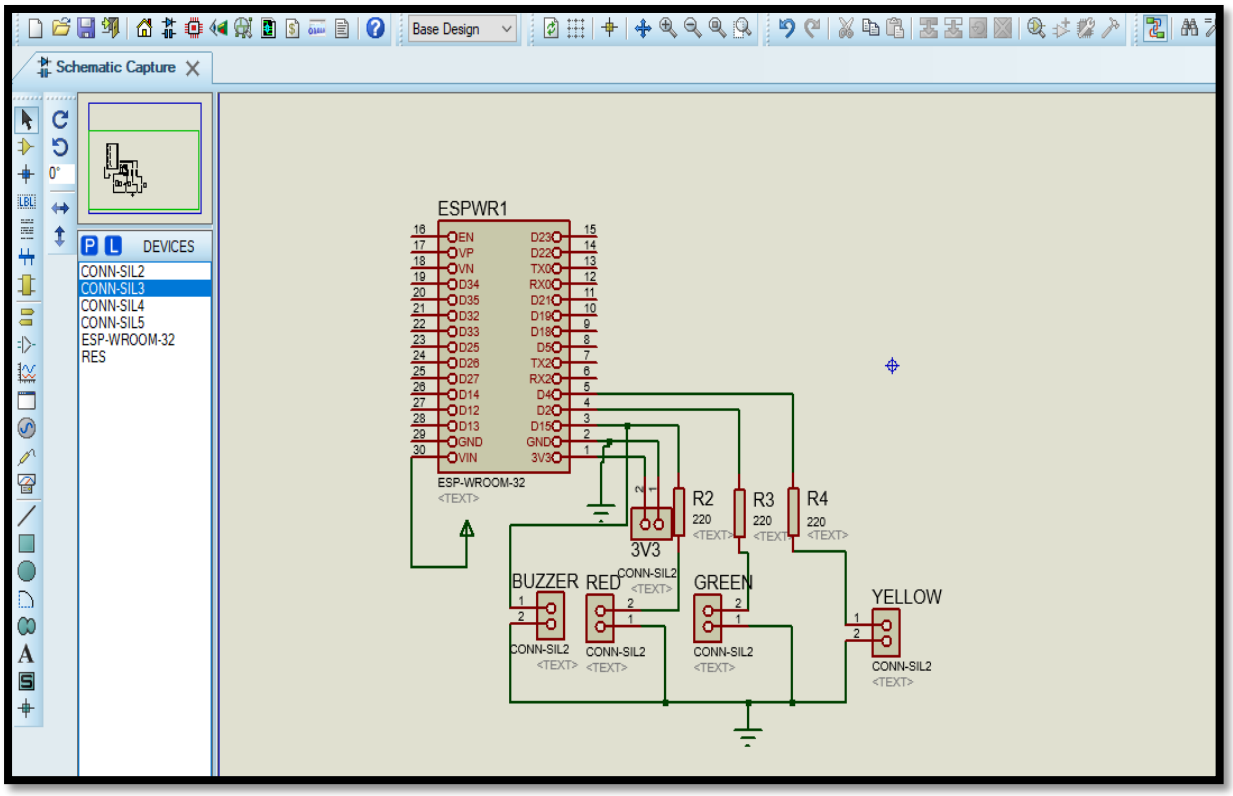


Figure 3.20: Buzzer and LED Configuration

Figure 3.20 shows how the 16x2 LCD display is used to show real-time battery parameters such as voltage, current, temperature and system status.

The status indicating LEDs are connected through a current limiting resistor of 220 Ω to ensure that excessive current does not flow through the LEDs, which could otherwise damage them or reduce their lifespan. The resistor limits the current to a safe level, typically around 10–20 mA, depending on the supply voltage.

3.4.7.2 System Alert Process Flow

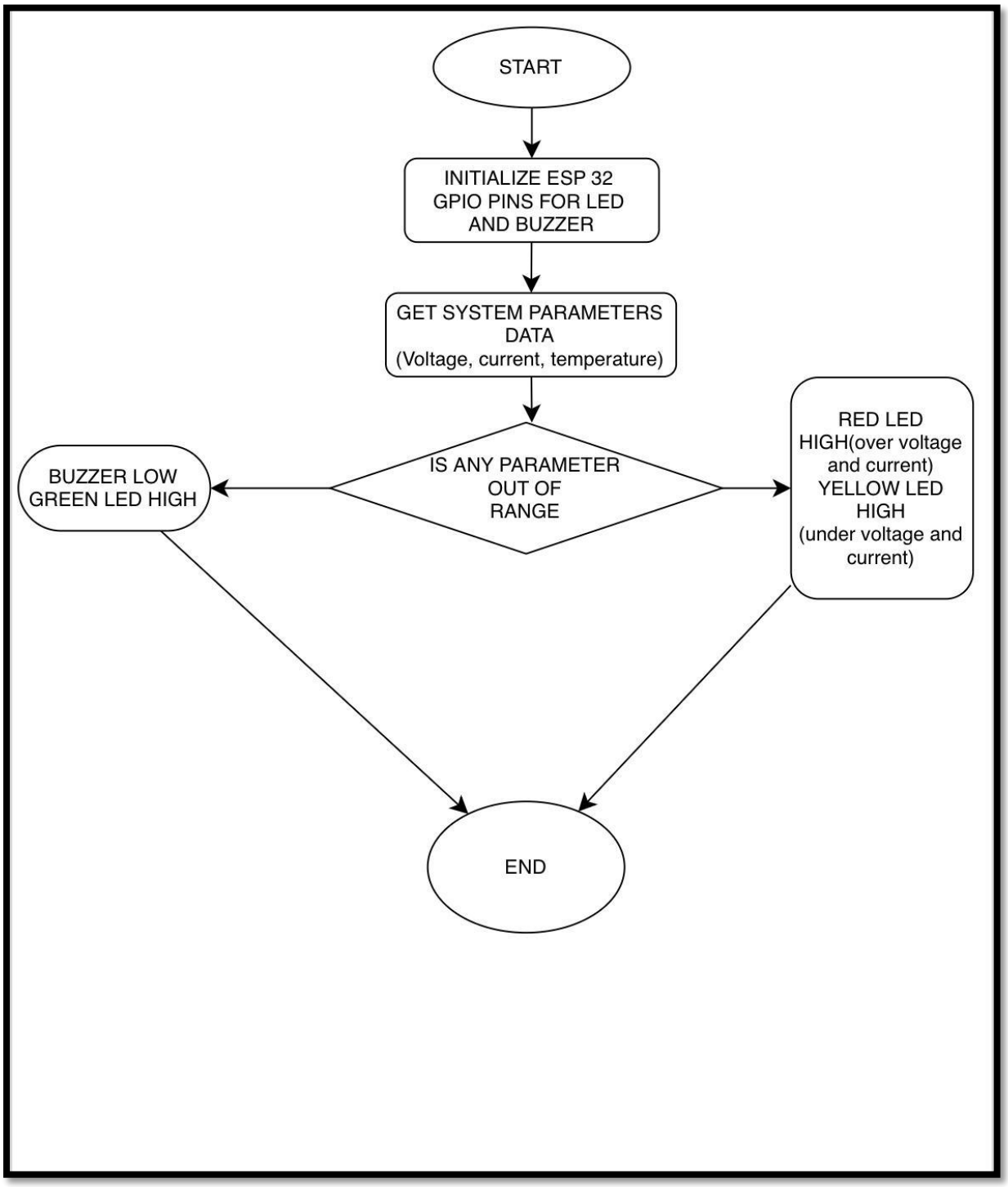


Figure 3.21: System Alert Process Flow

Figure 3.21 illustrates the system alert logic flowchart implemented in the Battery Management System (BMS) using an ESP32 microcontroller. The flowchart outlines how the system continuously monitors key electrical parameters and provides real-time feedback via LEDs and a buzzer based on the battery's health and safety status.

Process Description:

1. Start:
   The process begins with system initialization after power-up or reset.
2. Initialize ESP32 GPIO Pins:
   The ESP32 microcontroller sets up its General Purpose Input/Output (GPIO) pins configured to control the LED indicators and buzzer. These pins are prepared for digital output operations.
3. Get System Parameters (Voltage, Current, Temperature):
   The microcontroller reads the real-time system parameters using connected sensors. These readings are essential to determine the operating condition of the battery system.
4. Is Any Parameter Out of Range?
   The system compares the measured voltage, current, and temperature values against predefined safe thresholds.
   - If Parameters Are Out of Range:
     - The system turns ON the RED LED if over-voltage or over-current is detected.
     - The YELLOW LED turns ON if under-voltage or under-current conditions are present.
     - These visual alerts allow users to immediately identify abnormal conditions.
   - If All Parameters Are Normal:
     - The system turns ON the GREEN LED to indicate that all conditions are within safe limits.
     - The buzzer remains OFF, signaling normal operation.
5. End:
   This represents the end of the current checking cycle. In practice, this process loops continuously to ensure real-time system monitoring.

3.5 COMPLETE SYSTEM OPERATION FLOWCHART

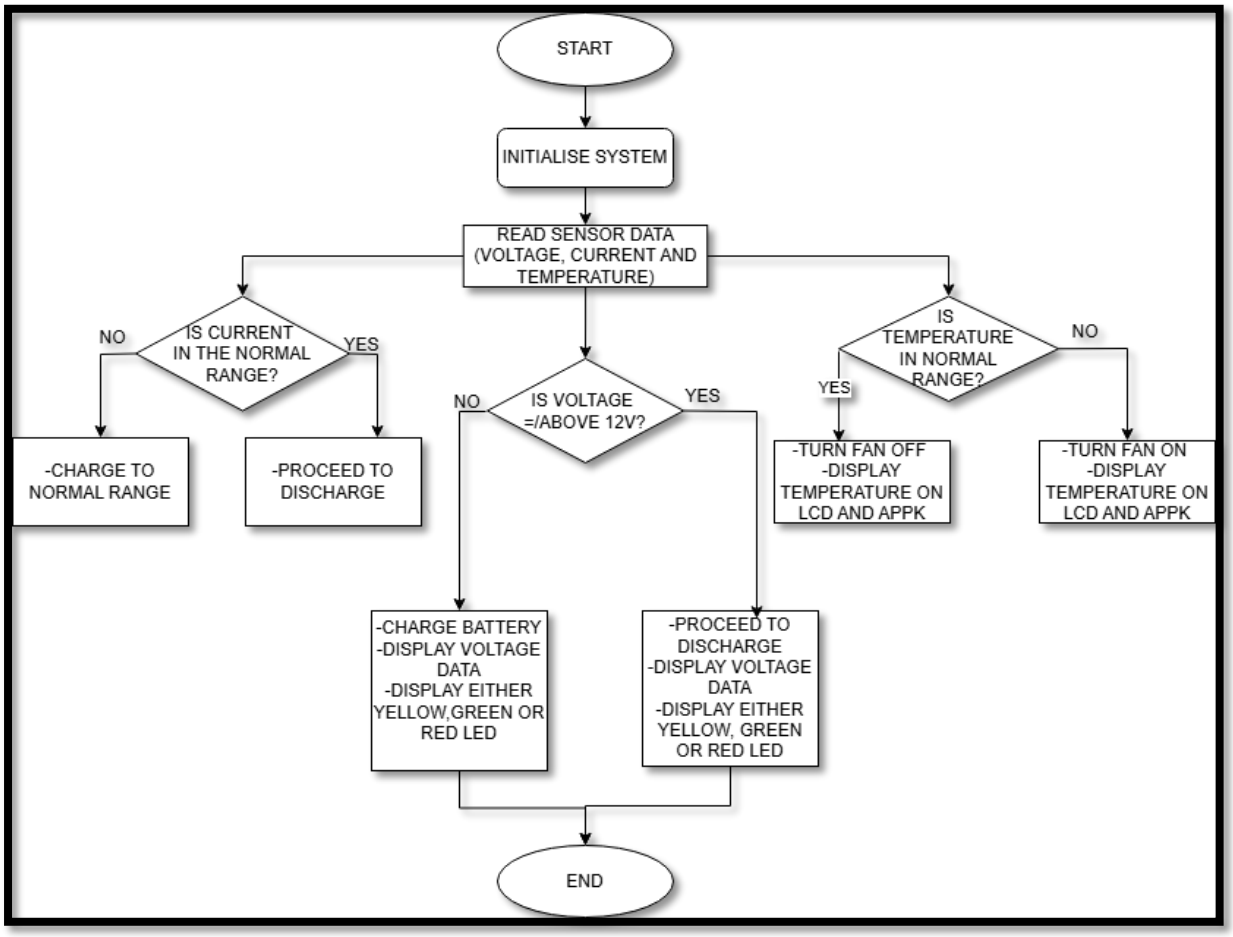


Figure 3.12: System Operation Flowchart

Figure 3.22 illustrates the complete operational flow of the Battery Management System (BMS). It outlines how the system manages the battery's voltage, current, and temperature in real-time by making intelligent decisions on charging, discharging, cooling, and user notifications through an LCD and mobile app interface.

Step-by-Step Explanation:

1. Start
   The system begins execution when it is powered ON or reset.
2. Initialize System
   The ESP32 microcontroller initializes all required peripherals and sensors including ADCs, temperature sensors, voltage/current sensors, cooling fan control, LCD, and communication with the Android application.
3. Read Sensor Data (Voltage, Current, Temperature)

   The system continuously collects real-time data from sensors that measure:

   - Battery voltage
   - Charge/discharge current

- Temperature of battery or environment

4. Is Current in the Normal Range?
   - If YES: The system continues to charge the battery as needed to maintain optimal performance.
   - If NO: The system interprets this as a potential overload or excessive current draw and shifts operation to discharge mode to protect the battery.
5. Is Voltage ≥ 100%?
   - If YES: The system decides to proceed to discharge (as full charge is reached).
     - Voltage level is displayed on the LCD screen and mobile app.
     - LED color status is updated (Green for full, Yellow for medium, Red for low).
   - If NO: The system initiates charging.
     - Voltage data is still monitored and displayed.
     - LED indicators provide charge level status using Green, Yellow, or Red.
6. Is Temperature in Normal Range?
   - If YES: The cooling fan is turned OFF to conserve energy.
   - If NO: The fan is activated to reduce battery temperature.
   - In both cases, temperature data is displayed on the LCD and transmitted to the Android application.
7. End
   The process loop ends and restarts immediately, forming a continuous monitoring and control system.

3.5.1 BMS PCB Layout Design

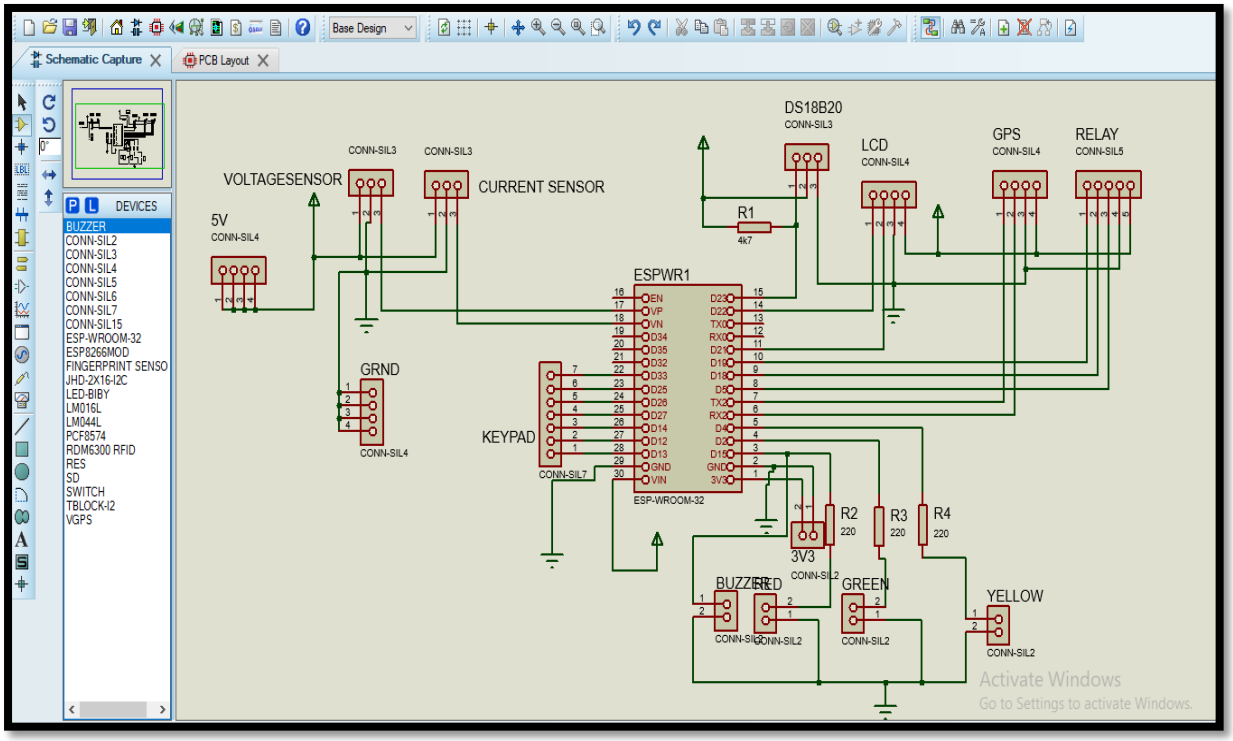


Figure 3.13: BMS PCB Layout Design

Figure 3.22 above shows the PCB Layout design of the Battery Management System (BMS).

It gives a detailed illustration on how various components are interconnected with the ESP32 microcontroller to enable real-time monitoring and control. The ESP32 serves as the central processing unit, responsible for collecting data from sensors and facilitating communication through Wi-Fi. It provides both 3.3V and 5V power supplies to different sensors based on their operating voltage requirements. The 25V voltage sensor is connected to GPIO34 (ADC1_CH6) of the ESP32, utilizing a voltage divider circuit to step down the battery voltage to a safe range before being read by the microcontroller. Similarly, the ACS712 current sensor is connected to GPIO35 (ADC1_CH7) and operates based on the Hall-effect principle, outputting an analogue voltage that corresponds to the measured current flow.

To monitor temperature, a DS18B20 digital temperature sensor is interfaced with GPIO 23 of the ESP32 using 1-wire communication, ensuring precise battery temperature readings to prevent overheating. For real-time tracking, the NEO-6M GPS module is integrated via UART (TX/RX), providing accurate location data to help with theft prevention and asset management. Security is further enhanced through a keypad authentication system, connected to designated ESP32 GPIOs, which restricts unauthorized access to battery settings.

A 16x2 LCD display, connected via the I2C protocol using GPIO21 (SDA) and GPIO22 (SCL), provides real-time visual feedback of battery parameters such as voltage, current, temperature, and status updates. Additionally, the system includes three LEDs, each representing different battery status levels. The green LED (connected to GPIO2) indicates a fully charged battery, the yellow LED (connected to GPIO4) warns of a low battery level, and the red LED (connected to GPIO15) signals critical battery discharge or fault conditions.

The ESP32 also transmits real-time battery parameters, including voltage, current, temperature, and location data, to an android application via Wi-Fi, allowing users to monitor system performance remotely. Additionally, the system is designed to trigger alerts in case of overvoltage, deep discharge or thermal runaway, ensuring proactive battery protection.

3.6 DEVELOPING THE ANDROID APPLICATION

Developing the android application for real time monitoring was done in two stages were a webpage was first developed using Xampp software and a later stage in which there was the conversion of the developed webpage into an android application using Ionic software. To convert the developed webpage into an Android application using Ionic, the process begins

with setting up the Ionic framework, which is a popular open-source toolkit for building cross-platform mobile applications using web technologies such as HTML, CSS, and JavaScript. Ionic allows the integration of web-based applications into a native Android app by leveraging Apache Cordova and Capacitor, enabling access to native device features.

3.6.1 Developing a Database for BMS Using XAMP

3.6.1.1 Introduction

The goal of this project is to develop a Battery Management System (BMS) to monitor and manage real-time battery data, including voltage, current, temperature, and GPS location. This system is built using XAMPP, MySQL, and PHP to store and manage the data effectively. The project involves creating a database, setting up tables for battery data and user authentication, and enabling communication between the ESP32 microcontroller and the database via a PHP API.

3.6.1.2 Database Setup

Installing XAMPP: XAMPP (Figure 3.17) was installed as the local server environment to manage the BMS database. XAMPP includes Apache for PHP processing, MySQL for database management, and phpMyAdmin for easy database administration. After installing XAMPP, the Apache and MySQL services were started to enable local database management.

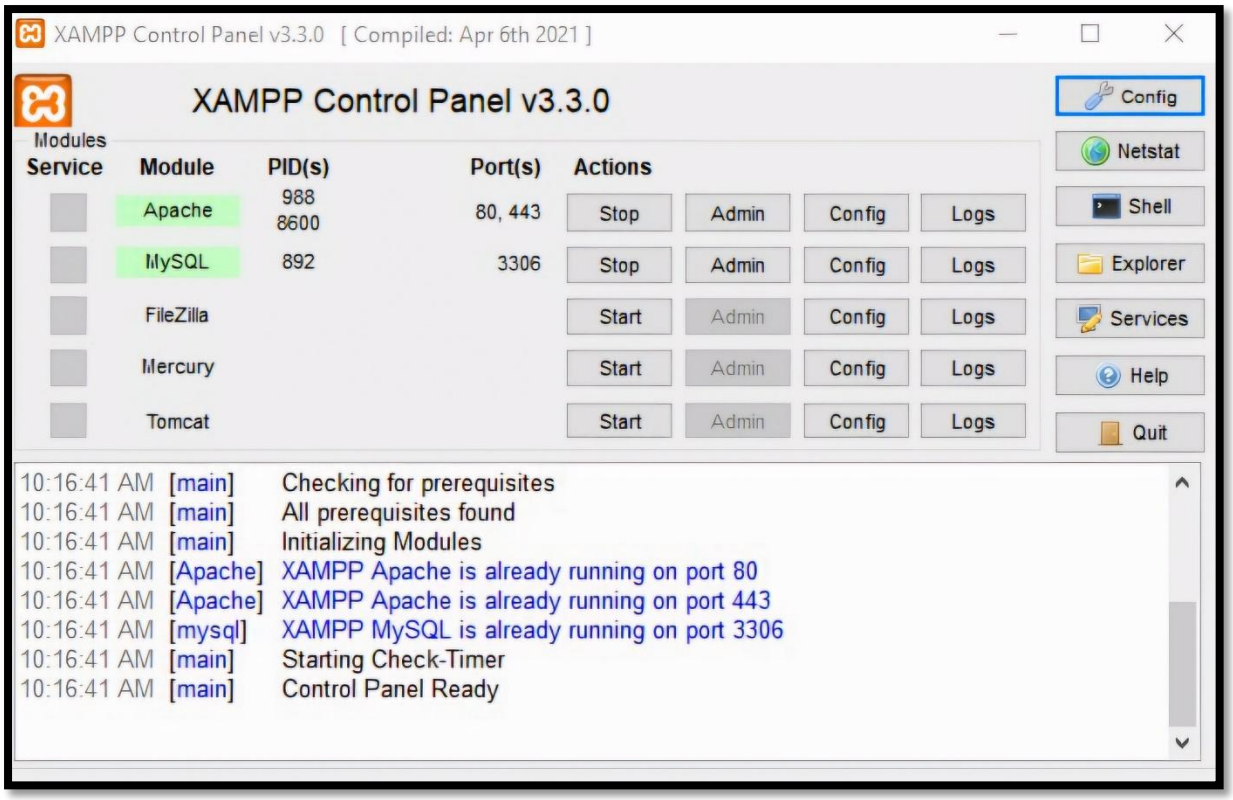


Figure 3.17: ***XAMPP Control Panel***

Creating the Database: In phpMyAdmin, a new database was created named bms_database. This database will store the battery data as well as user credentials for authentication. It was crucial to ensure that the database was properly set up before proceeding with the table creation.

Table Creation: Two primary tables were created to store relevant data:

- **Battery Data Table (battery_data)**: This table stores real-time battery parameters such as voltage, current, temperature, and GPS coordinates. The table also logs a timestamp for each data entry. The structure of the table is as follows:

Field Name	Data Type	Description
Id	INT (Primary Key, Auto Increment)	Unique entry ID
Voltage	FLOAT	Measured battery voltage
Current	FLOAT	Measured current flow
temperature	FLOAT	Battery temperature (°C)
gps_latitude	DOUBLE	Latitude value

Field Name	Data Type	Description
gps_longitude	DOUBLE	Longitude value
Timestamp	TIMESTAMP	Data logging time

3.6.1.3 Data Insertion and Testing

- **Inserting Sample Data:** To test the database, sample data was inserted into the tables:
- **Battery Data**: A sample record was inserted into the battery_data table, simulating typical sensor data values, such as a battery voltage of 12.5V, a current of 1.2A, a temperature of 28.4°C, and GPS coordinates for latitude and longitude.
- **User Authentication**: A sample user credential was inserted into the users table using the MD5 hashing algorithm for password storage, ensuring secure authentication.

3.7 Communication Between ESP32 and MySQL Database

PHP API for Data Insertion: Since the ESP32 cannot directly interact with MySQL, a PHP script (save_data.php) was created to handle the data insertion process. The PHP script (Figure 3.18):

- Accepts data from the ESP32 via HTTP POST requests (such as voltage, current, temperature, and GPS coordinates).
- Inserts this data into the battery_data table of the MySQL database.
- Provides feedback on whether the data insertion was successful or if there was an error

```
// Database connection parameters

$servername = "localhost";   // Hostname of the database server

$username = "root";          // MySQL username (default is root for localhost)
```

```
$password = "";             // MySQL password (leave empty for default XAMPP setup)

$dbname = "bms_database";    // Name of the database

// Create connection to MySQL database

$conn = new mysqli($servername, $username, $password, $dbname);

// Check the database connection

if ($conn->connect_error) {

   die("Connection failed: " . $conn->connect_error);

}

// Get data sent from ESP32 using HTTP POST method

$voltage = $_POST['voltage'];

$current = $_POST['current'];

$temperature = $_POST['temperature'];

$latitude = $_POST['gps_latitude'];

$longitude = $_POST['gps_longitude'];

// SQL query to insert data into the 'battery_data' table

$sql = "INSERT INTO battery_data (voltage, current, temperature, gps_latitude, gps_longitude)

      VALUES ('$voltage', '$current', '$temperature', '$latitude', '$longitude')";

// Execute the SQL query and check if it was successful

if ($conn->query($sql) === TRUE) {

   echo "Data stored successfully";
```

```
} else {
    echo "Error: " . $sql . "<br>" . $conn->error;
}
// Close the database connection
$conn->close();
```

3.7.1 ESP32 Program

The ESP32 is programmed to collect sensor data and send it to the PHP API using HTTP POST. The code on the ESP32 connects to a Wi-Fi network and periodically sends simulated sensor data to the PHP script running on the local server.

```
#include <WiFi.h>           // Library for WiFi connection
#include <HTTPClient.h>     // Library to send HTTP requests
// Replace with your actual Wi-Fi credentials
const char* ssid = "Your_WIFI_SSID";
const char* password = "Your_WIFI_Password";
// Server URL - Update with your local server IP or hosted PHP URL
const char* serverName = "http://192.168.1.100/bms/save_data.php"; // Change IP if using XAMPP/WAMP
void setup() {
  Serial.begin(115200);  // Start serial monitor for debugging
  // Connect to Wi-Fi
  WiFi.begin(ssid, password);
```

```
  while (WiFi.status() != WL_CONNECTED) {
    delay(1000);
    Serial.println("Connecting...");
  }
  Serial.println("WiFi Connected!");
}
void loop() {
  if (WiFi.status() == WL_CONNECTED) {
    HTTPClient http;
    http.begin(serverName);  // Set server URL
    http.addHeader("Content-Type", "application/x-www-form-urlencoded");    // Set HTTP
header


    // Simulated sensor data (replace with actual sensor readings)
    float voltage = 12.5;
    float current = 1.28;
    float temperature = 28.4;
    float gps_lat = -1.283689;
    float gps_lon = 36.817223;
    // Prepare the data to send (HTTP POST format)
    String postData = "voltage=" + String(voltage) +
```

```
                "&current=" + String(current) +
                "&temperature=" + String(temperature) +
                "&gps_latitude=" + String(gps_lat) +
                "&gps_longitude=" + String(gps_lon);
  // Send HTTP POST request
  int httpResponseCode = http.POST(postData);
  // Handle response
  if (httpResponseCode > 0) {
    Serial.println("Data Sent Successfully!");
} else {
    Serial.print("Error: ");
    Serial.println(httpResponseCode);
  }
  http.end();  // Close connection
 }
 delay(10000);  // Wait 10 seconds before sending again
}
```

3.8 Viewing and Managing Data

The data inserted into the database can be viewed and managed using phpMyAdmin. By navigating to http://localhost/phpmyadmin/, users can access the bms_database and view the

contents of the battery_data table, where sensor readings are logged. This enables easy monitoring and analysis of the battery parameters

3.9 Converting the BMS Webpage into an Android Application Using Ionic

This section outlines the process of converting an existing Battery Management System (BMS) monitoring webpage into a fully functional Android application using Ionic and Capacitor. The webpage, previously developed for real-time monitoring of battery data, will be packaged as a mobile app to allow users to monitor battery parameters such as voltage, current, temperature, and GPS location directly on Android devices.

3.9.1 Steps to Convert the Webpage into an Android App Using Ionic

3.9.1.1 Install Prerequisites

Before starting the development process, the following tools were installed:

- Node.js (which includes npm) for managing the app's dependencies.
- Ionic CLI to build and manage the app.
- Android Studio for testing and building APKs.
- Git (optional) for version control.

The installation process involved downloading Node.js from the official website

```bash
npm install -g @ionic/cli
```

Verifying the installation was done by checking the versions of Ionic, Node.js, and npm:

```bash
ionic -v   # Check Ionic version
node -v    # Check Node.js version
npm -v     # Check npm version
```

3.9.2 Creating an Ionic Project

To create a new Ionic project, the following command was used:

```bash
ionic start BMSApp blank --type=angular
```

Next, Capacitor was installed, a tool that enables running the web app as a mobile app.

```bash
npm install @capacitor/core @capacitor/cli
ionic build
npx cap init BMSApp com.example.bmsapp
```

3.9.3 Add Webpage Files to the Ionic Project

The default src/app/home folder in the Ionic project was deleted and replaced by the existing webpage files (HTML, CSS, and JavaScript) into the src. This ensured that the index.html file correctly referenced your webpage's files

<!DOCTYPE html>

<!-- This declares the document type as HTML5 -->

<html lang="en">

<!-- Root element of the HTML document, with language set to English -->

<head>

<meta charset="UTF-8">

<!-- Sets the character encoding to UTF-8 for proper text display -->

<meta name="viewport" content="width=device-width, initial-scale=1.0">

<!-- Ensures proper scaling on all devices, making the page responsive -->

<title>BMS Monitoring</title>

```
    <!-- Title of the webpage that appears in the browser tab -->
    <link rel="stylesheet" href="style.css">
    <!-- Links an external CSS file for styling the webpage -->
</head>
<body>
    <!-- The body of the webpage where visible content is placed -->
    <h1>Battery Management System</h1>
    <!-- Main heading displayed at the top of the page -->
    <div id="data-container">
        <!-- Container where data from ESP32 will be displayed -->
        <!-- Battery data from ESP32 will appear here -->
    </div>
    <script src="script.js"></script>
    <!-- Links an external JavaScript file to handle logic like fetching and displaying ESP32 data -->
</body>
</html>
```

3.9.4Test the App Using Ionic

To test the webpage as an app in the browser, the following command was used:

```
bash

ionic serve
```

This opened a live preview of your app, allowing you to verify that all functionality works as intended.

3.9.5 Add Android Platform

Since the goal is to convert the web application into an Android app, the next step was to add the Android platform using the following command:

```bash
ionic capacitor add android
```

After adding the platform, the Android project in Android Studio was opened:

```bash
npx cap open android
```

3.9.6 Configure Android Permissions

For use of features such as GPS, Wi-Fi and sensors, the appropriate permissions were added to the AndroidManifest.xml file. Here are some common permissions:

```xml
<uses-permission android:name="android.permission.INTERNET"/>
<uses-permission android:name="android.permission.ACCESS_FINE_LOCATION"/>
<uses-permission android:name="android.permission.ACCESS_NETWORK_STATE"/>
```

Once the permissions are added, the app was rebuilt with the following commands:

```bash
ionic build
npx cap sync android
```

3.9.7 Build the APK for Android Deployment

To generate the Android APK file for testing or deployment, navigate to the Android directory and run:

```bash
cd android
./gradlew assembleDebug
```

The APK will be generated at:

```swift
android/app/build/outputs/apk/debug/app-debug.apk
```

This APK file can be transferred and installed on a physical Android.

3.10 Conclusion

In conclusion, the system design of the Battery Management System (BMS) successfully integrated key components, including real-time monitoring of battery parameters such as voltage, current, temperature, and GPS location. The use of an ESP32 microcontroller, coupled with sensors like the ACS712, DS18B20, and NEO-6M, ensures accurate data collection. This data is efficiently managed and stored in a MySQL database via an API, while the system is accessible through an Android application created using Ionic and Capacitor. The design ensures seamless communication, secure access, and effective monitoring, providing a robust and user-friendly solution for battery management and monitoring.

CHAPTER 4: SYSTEM IMPLEMENTATION

4.1 Introduction

This chapter presents the practical implementation of the Battery Management System (BMS), translating the system design into a working prototype. It includes the integration of hardware components (sensors, microcontroller, keypad, LCD, GPS) and software development (firmware and mobile application). Each subsystem is tested and refined for real-time monitoring, fault detection, and secure access control.

4.2 System Integration

The ESP32 microcontroller serves as the core controller, interfacing with all sensors and modules. Each subsection below explains the integration and includes labeled placeholders for corresponding diagrams and code.

4.2 COMPLETE SYSTEM OPERATION FLOWCHART

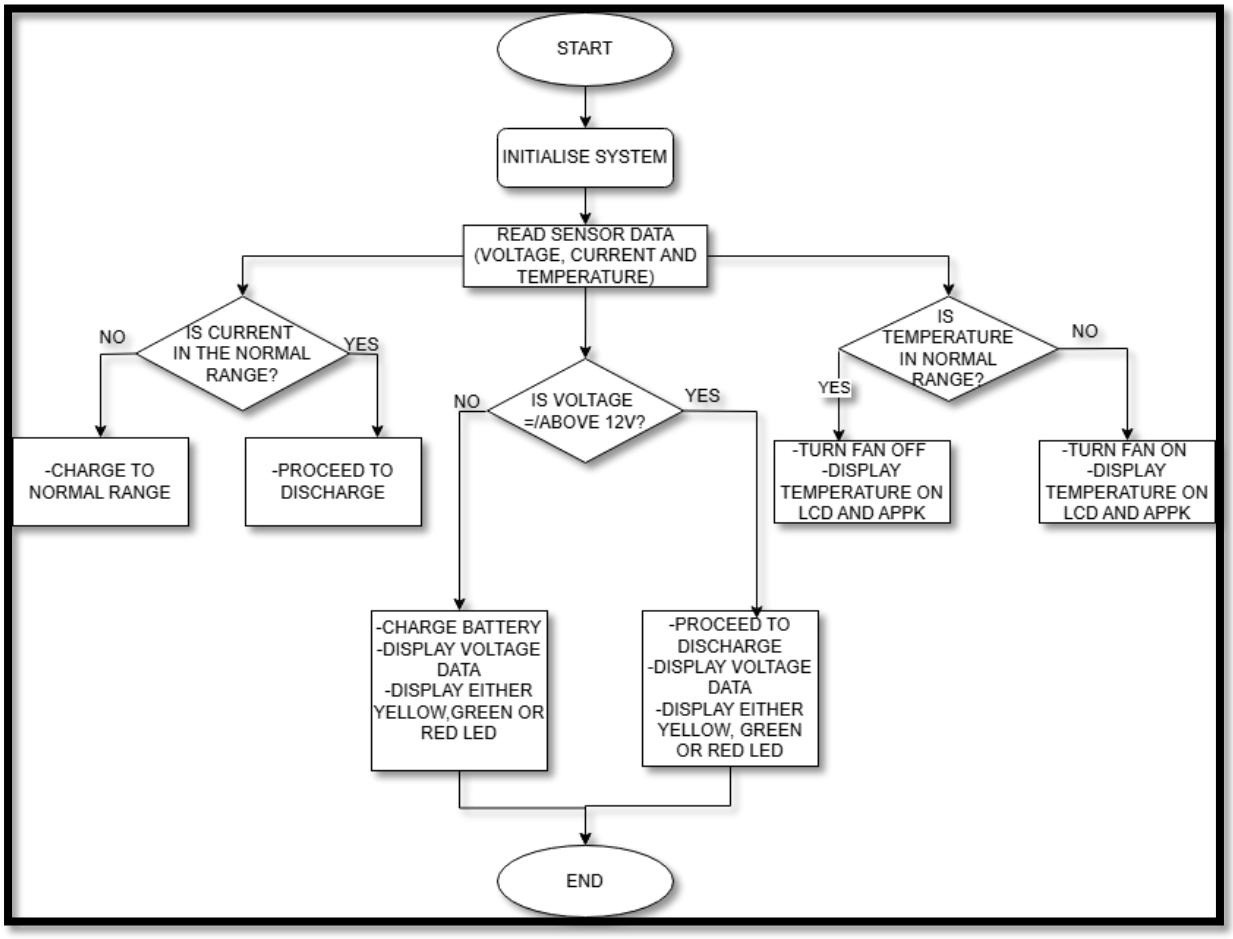


Figure 4.0: System Operation Flowchart

Figure 4.0 illustrates the complete operational flow of the Battery Management System (BMS). It outlines how the system manages the battery's voltage, current, and temperature in real-time by making intelligent decisions on charging, discharging, cooling, and user notifications through an LCD and mobile app interface.

Step-by-Step Explanation:

1. Start

   The system begins execution when it is powered ON or reset.

2. Initialize System

   The ESP32 microcontroller initializes all required peripherals and sensors including ADCs, temperature sensors, voltage/current sensors, cooling fan control, LCD, and communication with the Android application.

3. Read sensor data (voltage ,current ,temperature).
   - The system continuously collects real-time data from sensors that measure:
     - Battery voltage
     - Charge/discharge current
     - Temperature of battery or environment

4. Is Current in the Normal Range?
   - If YES: The system continues to charge the battery as needed to maintain optimal performance.
   - If NO: The system interprets this as a potential overload or excessive current draw and shifts operation to discharge mode to protect the battery.

5. Is Voltage ≥ 100%?
   - If YES: The system decides to proceed to discharge (as full charge is reached).
     - Voltage level is displayed on the LCD screen and mobile app.
     - LED color status is updated (Green for full, Yellow for medium, Red for low).
   - If NO: The system initiates charging.
     - Voltage data is still monitored and displayed.
     - LED indicators provide charge level status using Green, Yellow, or Red.

6. Is Temperature in Normal Range?
   - If YES: The cooling fan is turned OFF to conserve energy.
   - If NO: The fan is activated to reduce battery temperature.
   - In both cases, temperature data is displayed on the LCD and transmitted to the Android application.

7. End

   The process loop ends and restarts immediately, forming a continuous monitoring and control system.

4.2.1 Voltage Sensor Integration

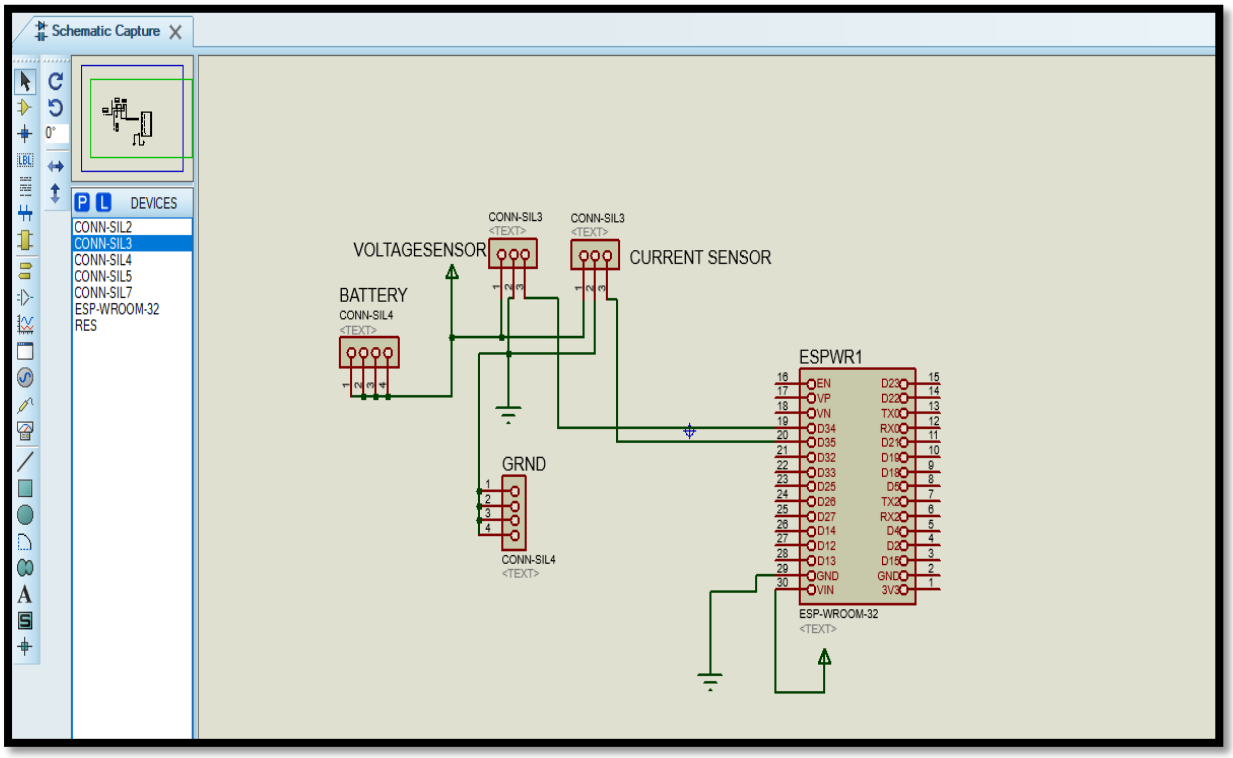


Figure 4.1: Voltage Sensor integration

The voltage sensing function of the Battery Management System was achieved using a 25V analogue voltage sensor, which was connected to analogue pin GPIO34 on the ESP32 microcontroller. This sensor internally uses a voltage divider network composed of precision resistors to safely reduce the high battery voltage to within the ESP32's 0–3.3V analogue input range. The ESP32 itself features a 12-bit Analog-to-Digital Converter (ADC), which converts analogue voltages into digital values ranging from 0 to 4095. For accurate monitoring, the relationship between the original battery voltage and the ADC value must be mathematically reconstructed in software using a known scaling factor.

```
int voltagePin = 34;            // Define analog input pin for voltage (ESP32 ADC pin)

float scaleFactor = 7.5;        // Scaling factor based on the voltage divider used

void setup() {

  Serial.begin(115200);         // Start serial communication for debugging

}
```

```
void loop() {
  int rawADC = analogRead(voltagePin);   // Read raw analog value from the voltage pin
  float voltage = rawADC * (3.3 / 4096.0) * scaleFactor;  // Convert ADC value to actual voltage
  Serial.print("Battery Voltage: ");     // Print label
  Serial.print(voltage);                 // Print calculated voltage
  Serial.println(" V");                  // Print voltage unit and move to next line
  delay(1000);   // Wait for 1 second before taking the next reading
}
```

The code above shows the program for voltage sensor integration. The code starts by declaring int voltagePin = 34, identifying the analogue pin that reads the voltage. A floating-point variable scaleFactor is initialized to 7.5, which corresponds to the inverse of the voltage division ratio. This means the actual input voltage is approximately 7.5 times the voltage measured at the analogue pin. In the setup () function, Serial.begin(115200) initializes communication with the Serial Monitor for real-time output.

Inside the loop () function, the command analogRead(voltagePin) captures the raw digital value from the ADC. This raw value is then converted to a real voltage using the expression rawADC * (3.3 / 4096.0), which maps the ADC reading to its corresponding voltage in the range of 0 to 3.3V. Since the voltage was reduced by the voltage divider, this measured voltage is then multiplied by scaleFactor to return to the actual battery voltage. The result is printed on the serial monitor and refreshed every second using delay (1000).

If the analogRead function returns a value of **1240**. The intermediate voltage at the analogue pin is calculated as:

- **Measured voltage =** $\frac{1\,240}{4\,096} \times 3.3$

  $= 1.0\ V$

Multiplying by the scale factor:

- **Actual battery voltage = 1.0 V×7.5**

  = 7.5 V

Thus, the real battery voltage in this example is approximately 7.5V. This dynamic calculation allows the ESP32 to accurately monitor the battery's charging and discharging behaviour in real time and can be used to trigger alerts when predefined voltage thresholds are crossed. This functionality is essential for implementing protections against deep discharge and overvoltage conditions.

4.2.2 Current Sensor Integration

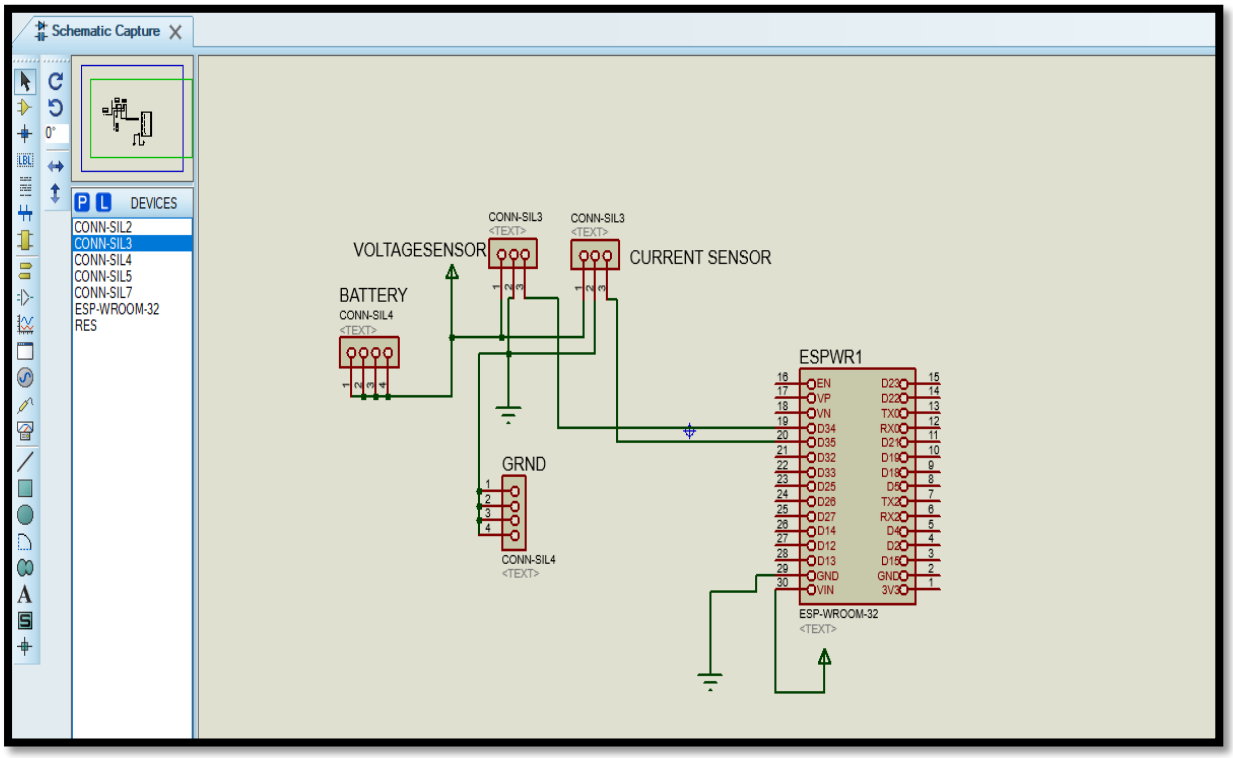


Figure 4.2: Current Sensor Configuration

Current monitoring was implemented using the ACS712 Hall-effect current sensor, which provides electrical isolation and analogue output proportional to the current flowing through it. In this system, the sensor's analogue output was connected to **GPIO35** on the ESP32 microcontroller. The ACS712 sensor has a **baseline voltage of 2.5V** when no current is flowing, and the output voltage shifts linearly in response to positive or negative current flow. The specific version used in the project was the **5A model**, which offers a sensitivity of **185 mV/A** meaning for every 1 ampere of current, the output voltage changes by 0.185 volts from the baseline. Here is the current integration code with ESP32:

```
// Define the analog input pin connected to the ACS712 current sensor
int currentPin = 35;
// Sensitivity of the ACS712 5A module (in V/A). For 5A, it's typically 185mV/A.
float sensitivity = 0.185;
void loop() {
  // Read the raw analog value from the sensor
  int rawValue = analogRead(currentPin);
  // Convert the raw analog value (0-4095) to voltage (0-3.3V for ESP32 ADC)
  float voltage = rawValue * (3.3 / 4096.0);
  // ACS712 outputs 2.5V at 0A. Subtract this midpoint to get the voltage difference.
  // Then divide by sensitivity to convert the voltage difference to current (in amperes).
  float current = (voltage - 2.5) / sensitivity;
  // Print the current value to the Serial Monitor
  Serial.print("Current: ");
 Serial.print(current);
  Serial.println(" A");
  // Wait for 1 second before the next reading
  delay(1000);
}
```

In the code, int currentPin = 35 designates the analogue input used for reading the sensor's output. The variable sensitivity is defined as 0.185 to match the 185 mV/A characteristic. The

setup () function initializes serial communication with the Serial.begin(115200) command. Inside the loop (), the function analogRead(currentPin) captures the raw ADC value from the ESP32's 12-bit analogue input (0–4095). This raw value is then converted into a voltage using rawValue * (3.3 / 4096.0), since the ESP32 operates with a 3.3V reference.

Because the ACS712 outputs 2.5V at zero current, the code subtracts 2.5 from the measured voltage to calculate the voltage offset caused by actual current flow. Dividing this result by the sensor's sensitivity gives the measured current in amperes. The current value is printed to the serial monitor and updated every second using delay (1000). This design enables continuous, real-time tracking of charging or discharging current levels.

Sample Calculation:

Consider an ADC return value of 3062. First, the raw ADC value is converted to voltage:

$$\textbf{Voltage} = \frac{3\,062}{4\,096} \times 3.3$$

$$= 2.47\ \text{V}$$

- Subtracting the baseline (2.5V):

$$\textbf{Offset Voltage} = 2.47 - 2.5$$

$$= -0.03\ \text{V}$$

- Dividing by sensitivity to get current:

$$\textbf{Current} = \frac{-0.03}{0.185}$$

$$= 0.16\ \text{A}$$

This negative value indicates reverse current flow, which is typical in charging scenarios. The real-time current monitoring is critical for detecting overcurrent faults and managing load activation or isolation accordingly.

4.2.3 Temperature Sensor Integration

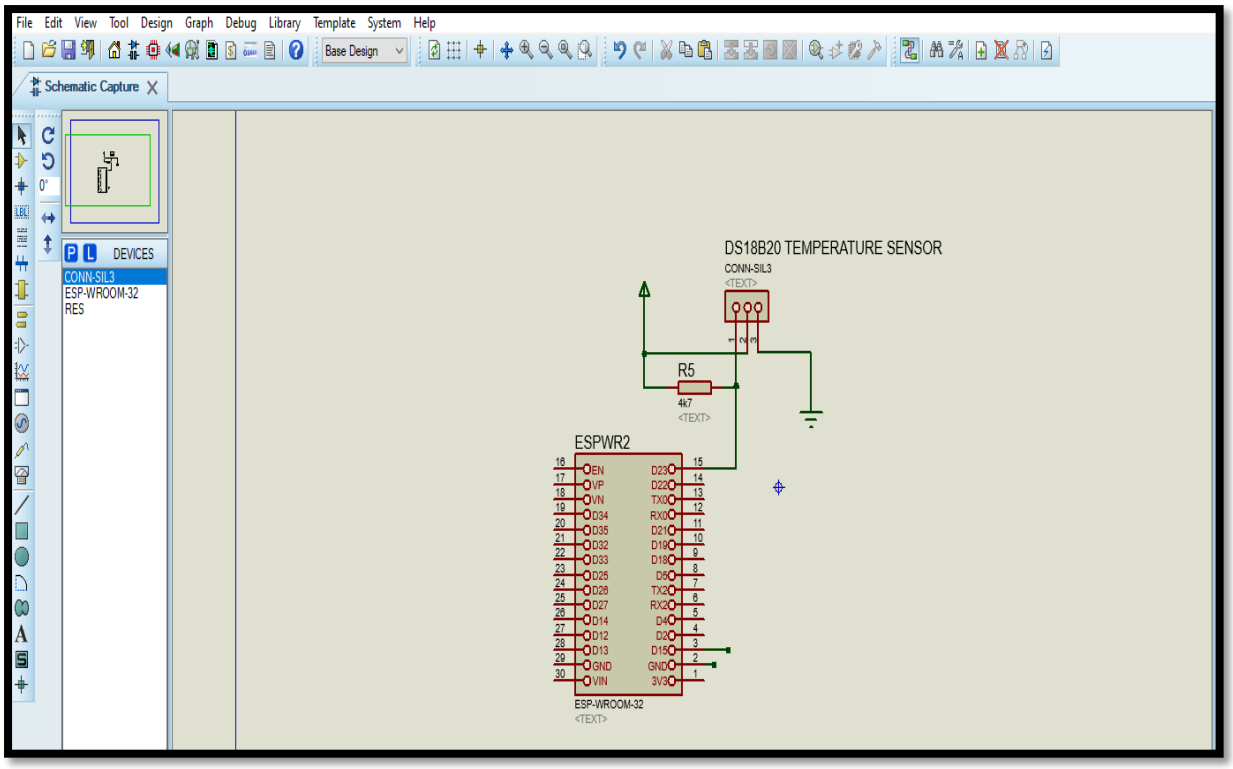


Figure 4.3: Temperature Sensor Circuit

The DS18B20 digital temperature sensor was employed in the BMS due to its high accuracy, ease of communication, and ability to support multiple devices on a single data bus using the OneWire protocol. The sensor was connected to **GPIO14** on the ESP32, with a **4.7kΩ pull-up resistor** placed between the data line and 3.3V to ensure signal integrity. The integration uses two libraries: OneWire.h for handling the low-level communication and DallasTemperature.h for higher-level sensor commands. The #define ONE_WIRE_BUS 14 line declares the GPIO pin being used, and the OneWire and DallasTemperature objects are initialized to enable interaction with the sensor. The temperature sensor configuration integration code is shown below:

```
// Include the libraries needed to communicate with the temperature sensor

#include <OneWire.h>

#include <DallasTemperature.h>

// Define the pin to which the DS18B20 data line is connected

#define ONE_WIRE_BUS 14
```

```
// Setup a OneWire instance to communicate with any OneWire device

OneWire oneWire(ONE_WIRE_BUS);

// Pass the OneWire reference to DallasTemperature library

DallasTemperature sensors(&oneWire);

void setup() {

  Serial.begin(115200);       // Start serial communication at 115200 baud rate

  sensors.begin();            // Initialize the temperature sensor

}

void loop() {

  sensors.requestTemperatures();               // Request temperature reading from the sensor

  float tempC = sensors.getTempCByIndex(0);      // Get temperature in Celsius from the first sensor on the bus

  Serial.print("Temperature: ");

  Serial.print(tempC);                          // Print temperature value

 Serial.println(" °C");                        // Print unit

  delay(1000);                                 // Wait 1 second before the next reading

}
```

In the setup () function, Serial.begin(115200) starts serial communication for debugging purposes, and sensors.begin () initializes the communication with any DS18B20 sensors detected on the OneWire bus. Within the loop (), sensors.requestTemperatures () sends a command to retrieve the latest temperature reading from the sensor. The value is then fetched

using sensors.getTempCByIndex (0), where 0 refers to the index of the first sensor on the bus (supporting scalability for multiple sensors). The temperature, returned as a float, is printed to the serial monitor in degrees Celsius, and the loop refreshes every second.

Unlike analogue sensors, the DS18B20 provides fully digital output**,** removing the need for ADC conversion and simplifying noise immunity. It also supports a precision of ±0.5°C over much of its range**,** making it suitable for thermal safety monitoring in lithium-based battery systems. The ESP32 continuously logs these readings and can trigger cooling actions or system shutdowns if thermal thresholds are breached.

Sample Output Interpretation:

Assuming sensors.getTempCByIndex(0) returns 36.25, this means the temperature at the sensor's position is **36.25°C**, a typical operational temperature under moderate battery usage. If the value exceeds a defined safety limit, 3**0°C**, the BMS can automatically cut off charging or alert the user to thermal stress, preventing thermal runaway conditions.

4.2.4 GPS Module Integration

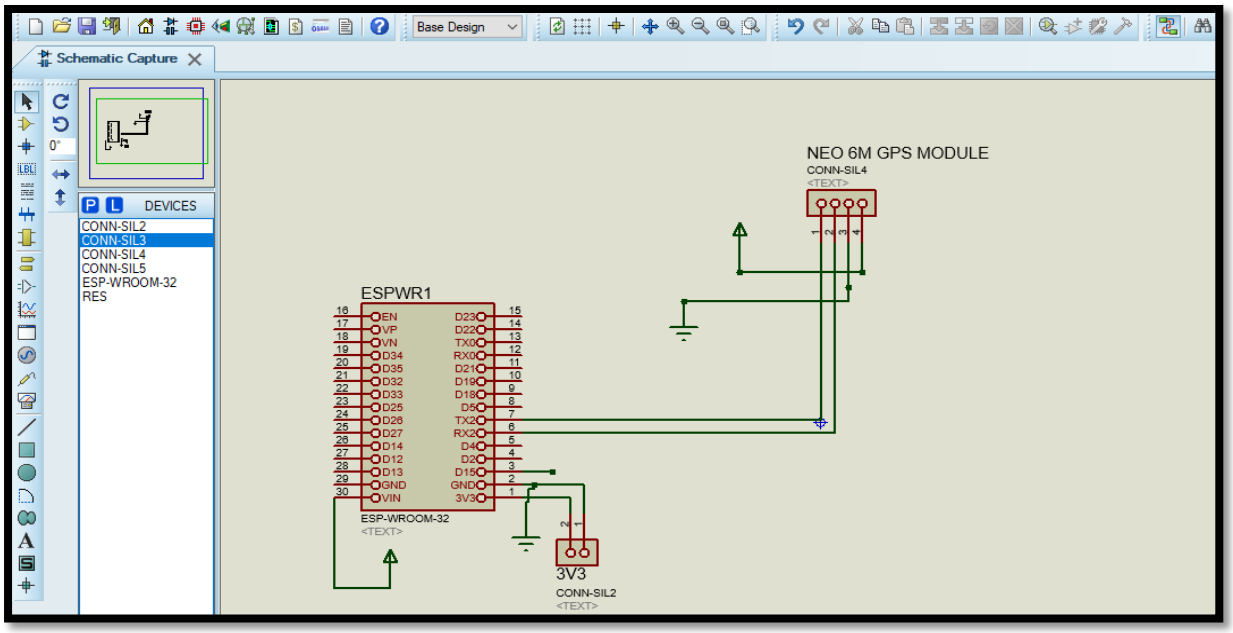


Figure 4.4: GPS Module Integration

The NEO-6M GPS module was integrated into the system to provide real-time geolocation data, which is essential for asset management and anti-theft functionality within the Battery

Management System. The module communicates with the ESP32 using UART2, with its TX pin connected to GPIO16 and RX to GPIO17**.** The TinyGPSPlus library is used for parsing raw NMEA sentences received from the module into human-readable location data. HardwareSerial(2) creates a second serial interface (Serial2), allowing the ESP32 to manage multiple serial connections simultaneously. The GPS module integration sensor configuration integration code is shown below:

```
#include <TinyGPS++.h>          // Include TinyGPS++ library for parsing GPS data
#include <HardwareSerial.h>     // Include HardwareSerial library to use Serial2
TinyGPSPlus gps;                // Create a TinyGPS++ object to handle GPS data
HardwareSerial ss(2);           // Create a second serial port (Serial2) on ESP32
void setup() {
  Serial.begin(115200);         // Start serial communication with the Serial Monitor
  ss.begin(9600, SERIAL_8N1, 16, 17);  // Start GPS serial communication on TX = 17, RX = 16
}
void loop() {
  // Read data from the GPS module and feed it to the TinyGPS++ library
  while (ss.available()) {
    gps.encode(ss.read());      // Parse incoming bytes from GPS module
  }
  // If new location data is available, print it
  if (gps.location.isUpdated()) {
    Serial.print("Latitude: ");
    Serial.println(gps.location.lat(), 6);    // Print latitude with 6 decimal places
    Serial.print("Longitude: ");
    Serial.println(gps.location.lng(), 6);    // Print longitude with 6 decimal places
  }
```

```
  delay(1000);  // Wait for 1 second before next read
}
```

In the setup() function, Serial.begin(115200) initializes the primary serial port for monitoring, while ss.begin(9600, SERIAL_8N1, 16, 17) initializes UART2 with standard 8-bit data, no parity, and 1 stop bit configuration. This setup ensures communication between the ESP32 and the GPS module at the default baud rate of 9600 bps.

Within the loop (), the while (ss.available() > 0) statement checks if there is incoming data from the GPS. If data is present, gps.encode (ss.read()) is used to decode the raw characters. Once a valid GPS frame has been received and decoded, gps.location.isUpdated() returns true, and the system prints the updated latitude and longitude values using gps.location.lat() and gps.location.lng(). The 6 in the print function specifies that the values should be printed with six decimal places of precision, ensuring high accuracy. The delay of one second at the end of the loop throttles the updates for readability and stability.

This GPS integration allows the ESP32 to push location data to a web server and Android application. The coordinates can be visualized using tools like Leaflet.js on the web or mobile dashboards, enabling live tracking of battery location. It also enables geofencing and location-based alerts if the battery is moved outside designated zones.

Sample Output:

Latitude: -17.829222
Longitude: 31.052514

This indicates the precise geographic position of the battery at a specific time, with accuracy down to a few meters. These coordinates are useful for recovery in case of theft or unauthorized movement and help manage distributed energy storage units across a wide area.

4.2.5 LCD Display Integration

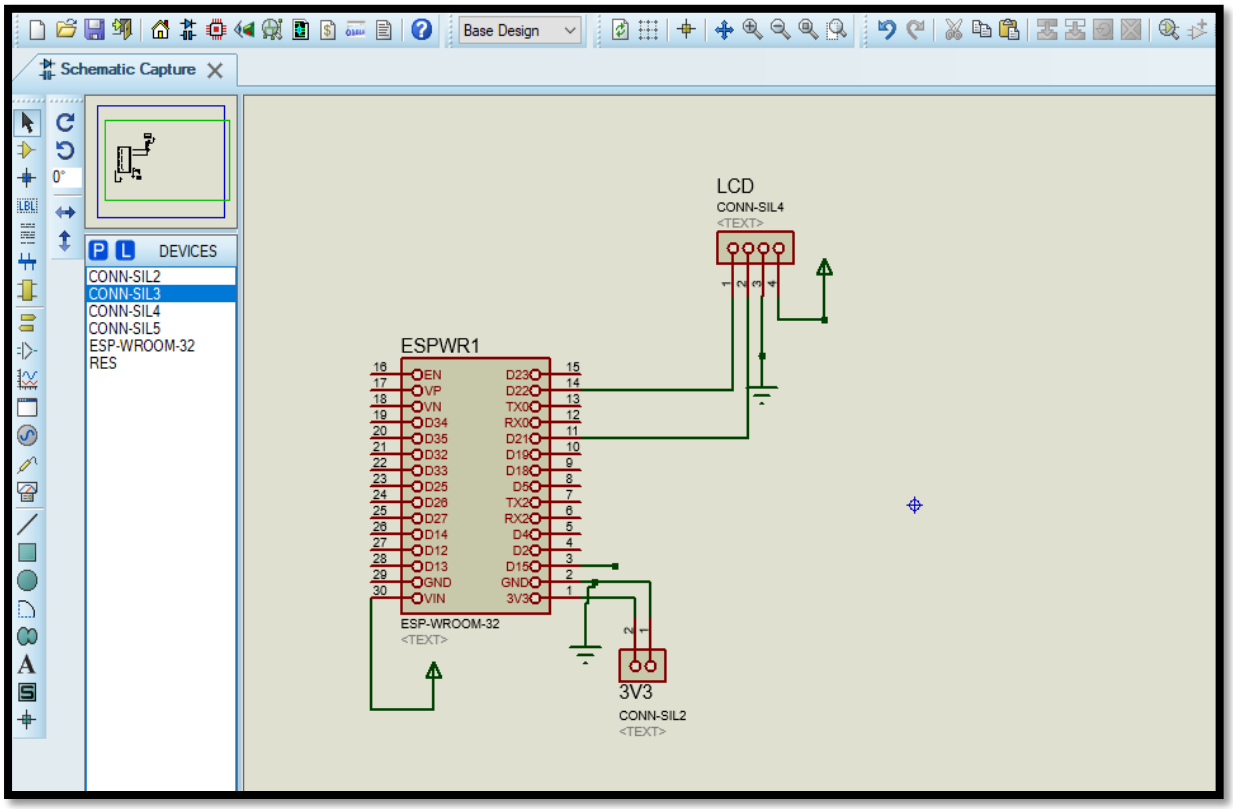


Figure 4.5: LCD Configuration via 12C

To provide a user-friendly visual interface for local monitoring, a 16x2 liquid crystal display (LCD) was integrated into the Battery Management System using the I2C communication protocol. This approach significantly reduced the number of wiring connections required, as only two GPIO pins on the ESP32 were needed: GPIO21 for the data line (SDA) and GPIO22 for the clock line (SCL). The display module was powered via the 5V supply rail, and an I2C backpack module was soldered to the rear of the LCD to facilitate I2C communication. This module has a default I2C address of 0x27, although other addresses may be used depending on the manufacturer. The LCD configuration integration code is shown below:

```
#include <Wire.h>                    // Include Wire library for I2C communication
#include <LiquidCrystal_I2C.h>       // Include LCD library for I2C LCD
// Initialize the LCD with I2C address 0x27, 16 columns and 2 rows
LiquidCrystal_I2C lcd(0x27, 16, 2);
void setup() {
```

```
  lcd.begin();                    // Initialize the LCD
  lcd.backlight();                // Turn on the LCD backlight
  lcd.setCursor(0, 0);            // Set cursor to column 0, row 0
  lcd.print("Battery BMS");       // Print message on the first line
}
void loop() {
  lcd.setCursor(0, 1);            // Set cursor to column 0, row 1
  lcd.print("Volt: 12.6V");       // Print voltage value (static example)
  delay(2000);                    // Wait for 2 seconds
}
```

The Arduino code includes the Wire and LiquidCrystal_I2C libraries, which handle I2C communication and text display, respectively. The display is initialized with LiquidCrystal_I2C lcd (0x27, 16, 2) to specify the I2C address and the display dimensions. In the setup function, lcd.begin() initializes the LCD, and lcd.backlight() ensures the display is visible. Initial messages are printed to the screen using lcd.setCursor() to set the starting column and row, followed by lcd.print() to display specific values or messages.

In the main loop, real-time parameters such as voltage, current, and temperature are dynamically displayed. For example, the LCD could be updated to show “Volt: 12.6V” or “Temp: 32.5°C” by setting the cursor to the appropriate row and calling lcd.print() with the sensor values. A delay or a timer-based refresh ensures that the data updates at a readable rate without flickering or data overlap.

Using the I2C interface reduces complexity and allows other I2C devices such as EEPROMs or additional sensors to be connected to the same communication lines, thus optimizing the ESP32’s limited GPIO availability. The LCD provides immediate visual feedback and is especially helpful in standalone or offline scenarios where mobile app or web-based monitoring is unavailable.

In this system, the loop continuously updates to show current, temperature, and status indicators in rotation or based on system events such as password entry success, GPS location availability, or sensor fault alerts.

4.2.6 Keypad module integration

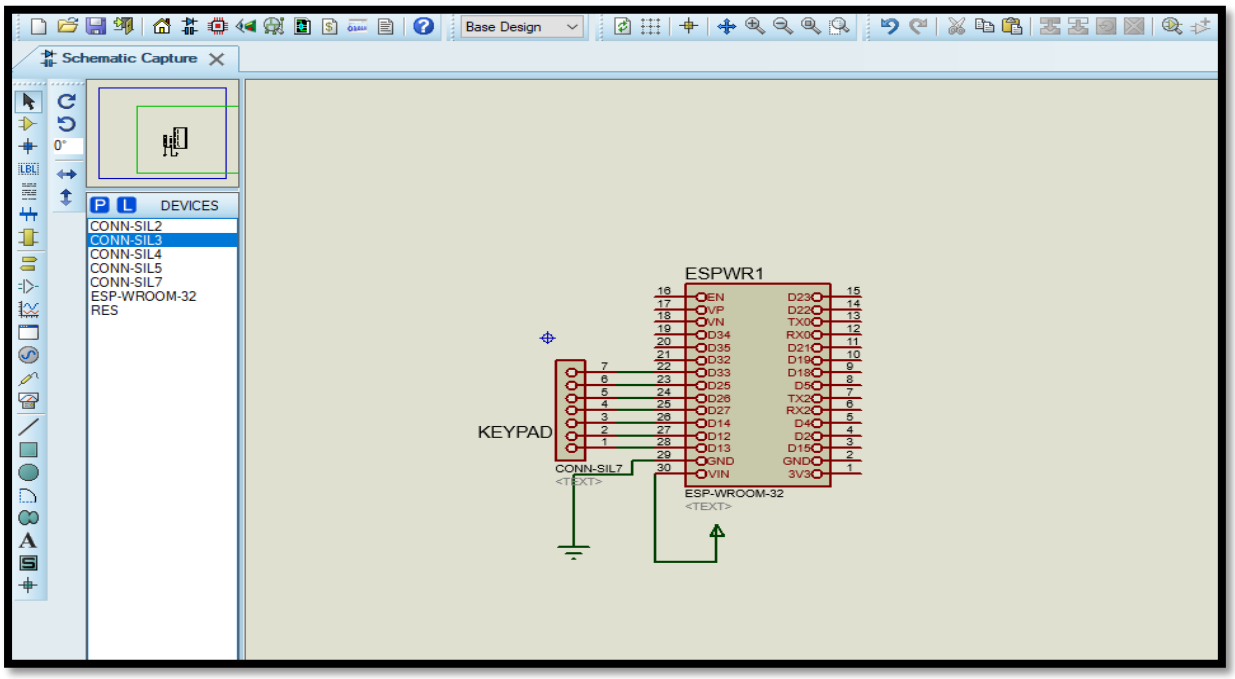


Figure 4.6: Keypad Module Configuration

To enhance the security of the Battery Management System, a 4x4 matrix keypad was integrated to serve as a manual authentication interface. This feature ensures that only authorized users can activate the battery output terminals by entering a predefined password. The keypad connects to eight GPIO pins on the ESP32 four for rows and four for columns. These digital pins continuously scan for key presses by detecting which row and column line intersect when a button is pressed. The user must input a four-digit code which is compared to a preset password stored in the program. If the code is correct, a relay connected to GPIO23 is activated to allow current flow from the battery. If the password is incorrect, the system locks the output by keeping the relay open. The keypad module configuration integration code is shown below:

```
#include <Keypad.h>  // Required for keypad input

// Define the correct password
```

```
String correctPassword = "1234";

// Define the relay control pin

int relayPin = 23;

// Variable to store the entered password

String inputPassword = "";

// Define keypad layout

const byte ROWS = 4; // Four rows

const byte COLS = 4; // Four columns

char keys[ROWS][COLS] = {

  {'1','2','3','A'},

  {'4','5','6','B'},

  {'7','8','9','C'},

  {'*','0','#','D'}

};

byte rowPins[ROWS] = {32, 33, 25, 26}; // Connect to the row pinouts of the keypad

byte colPins[COLS] = {27, 14, 12, 13}; // Connect to the column pinouts of the keypad

Keypad keypad = Keypad(makeKeymap(keys), rowPins, colPins, ROWS, COLS);

void setup() {

  pinMode(relayPin, OUTPUT);       // Set the relay pin as output

  digitalWrite(relayPin, LOW);     // Make sure relay is initially OFF
```

```
  Serial.begin(115200);           // Start serial communication
}
void loop() {
  char key = keypad.getKey();      // Read a key from the keypad
  if (key) {
    inputPassword += key;          // Append the pressed key to the password string
    Serial.println(inputPassword); // Show current input
  }
  if (inputPassword.length() == 4) {              // Once 4 characters are entered
    if (inputPassword == correctPassword) {       // If password matches
      digitalWrite(relayPin, HIGH);               // Activate relay
      Serial.println("Access Granted");
    } else {
      digitalWrite(relayPin, LOW);                // Keep relay OFF
      Serial.println("Access Denied");
    }
    inputPassword = "";                           // Reset for next input
  }
}
```

This program code is for acquiring key press data. The code begins by including the Keypad.h library and defining the keypad layout as a 4x4 grid of characters. The pins for rows and

columns are assigned to GPIOs 4–11. The Keypad object is then initialized with the defined keymap and pin configurations. A string variable input Password is used to store the user's entered digits, while correctPassword holds the expected 4-digit code, such as "1234." The relay pin is configured as an output in the setup function and initialized to LOW, meaning it is off by default.

Inside the loop, keypad.getKey() checks whether a button has been pressed. Each detected character is appended to inputPassword. Once four characters have been entered, the system compares this input to the correct password. If the values match, the system prints "Access Granted" and sets GPIO23 HIGH to activate the relay. Otherwise, it prints "Access Denied" and ensures the relay remains off. In both cases, inputPassword is reset to allow for a new entry.

This simple yet effective security feature adds a layer of protection to the battery, ensuring that only approved personnel can initiate discharge or access system functions. It is especially valuable in standalone or field-deployed energy systems where physical tampering or theft is a concern.

4.3 BMS PCB Layout Design

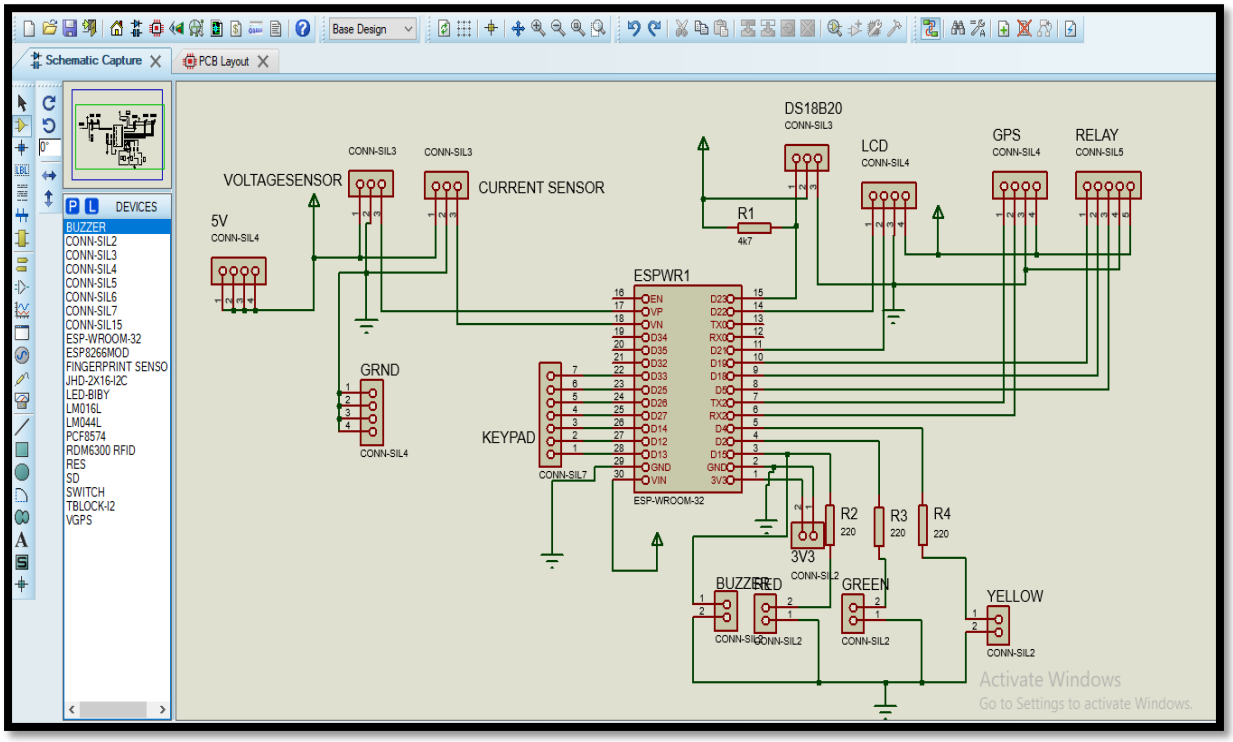


Figure 4.7: BMS PCB Layout Design

The figure 4.7 presents a full PCB layout design of a battery management and monitoring system designed using Proteus Design Suite. This system is developed to continuously track and manage vital battery parameters to ensure operational safety, reliability, and extended battery lifespan. The architecture is built around a central microcontroller, which serves as the brain of the system. It processes sensor inputs and executes control logic to drive various outputs such as display readings, status indicators, and relay operations. The system mimics a real-world embedded monitoring and control unit suitable for applications in solar power systems, electric vehicles, uninterruptible power supplies (UPS), and IoT-based power solutions.

At the heart of the system lies the microcontroller unit, which interfaces with multiple analogue and digital peripherals. It acquires analogue voltage and current values via ADC (Analog-to-Digital Conversion) channels and reads temperature data from an attached sensor. Based on the captured values, the microcontroller calculates key parameters such as battery voltage, charge percentage, temperature, and current, and relays this information to a 16x2 LCD display. The LCD, positioned centrally on the left side of the diagram, is configured to show real-time values in two rows. The first row reads “B: 9.56V 66.80%”, where “B” stands for battery

voltage measured at 9.56 volts and “66.80%” denotes the estimated state of charge (SOC) based on the voltage range and capacity. The second row, “C: 36°C 1.00A”, displays the battery temperature at 36 degrees Celsius and current measurement of 1.00 amperes, indicating either the load draws or charging current depending on system mode.

To sense the battery voltage, a voltage divider network is used (consisting of variable resistor RV1 and fixed resistors), which scales down the input battery voltage to a safe range readable by the microcontroller's ADC. For current measurement, the system likely incorporates a Hall-effect current sensor such as the ACS712, known for its high accuracy and electrical isolation. Alternatively, it may use a precision shunt resistor configuration for detecting current via voltage drop. The temperature is monitored using a sensor like LM35 or DS18B20, which provides a linear voltage output or digital reading proportional to temperature. These sensors provide essential input to monitor battery health and detect potential issues such as overheating or overcurrent situations.

The system also features integrated GPS and Wi-Fi communication modules, which are connected via UART ports labelled “GPS” and “WIFI” on the diagram. These modules allow for remote tracking of the device’s location and wireless data transmission, making the system IoT-enabled and suitable for remote diagnostics or asset tracking applications. These ports, along with the serial RX and TX communication lines, ensure that the system can interface with external modules or cloud platforms for advanced analytics and control.

Three coloured LED indicators red, yellow, and green are placed centrally to provide at-a-glance status monitoring. The green LED indicates a healthy battery, the yellow warns of a medium or borderline condition, and the red LED signals a low charge, fault, or potential system shutdown. This visual feedback mechanism enhances usability by allowing users to quickly interpret battery status without needing to read the LCD display.

Control of external devices, such as a pump or motor, is handled through a relay and transistor driver circuit located on the right side of the schematic. This section is crucial for load management. The microcontroller activates relays using switching transistors (BC547), which in turn control high-power devices without overloading the MCU. Fly back diodes are placed across relay coils to suppress inductive voltage spikes generated during switching, thereby protecting the transistors and improving system longevity. The pump motor symbol indicates

a typical use case, where the battery supplies power to a load such as a water pump in an irrigation system or a backup generator in a remote installation.

A standard LM7805 voltage regulator circuit is included at the top centre of the schematic to step down a higher input battery voltage (12V) to a regulated 5V required by the logic circuits and microcontroller. It includes filter capacitors for stability and noise reduction, ensuring clean power supply for sensitive electronics.

In addition, the system includes manual push buttons for resetting the microcontroller or toggling modes such as calibration or test. These buttons are connected with pull-up resistors to avoid floating inputs, ensuring the MCU reliably detects logic high or low states when pressed.

Overall, this circuit design illustrates a robust and intelligent embedded system for real-time battery monitoring and control. It demonstrates a holistic approach by combining sensing, data processing, visual display, relay actuation, and communication. With such a setup, the system can autonomously manage battery operation, warn users of unsafe conditions, and respond with appropriate control actions, making it highly suitable for deployment in both consumer and industrial energy applications.

4.4 System Testing

System testing was a critical phase in the implementation of the Battery Management System (BMS), serving to validate the accuracy, stability, and reliability of both hardware and software components. This section describes the step-by-step procedures undertaken to test individual modules, the integrated system, fault responses, and mobile application functionality. Through simulated and real-world conditions, the system was evaluated for its performance under normal operation and exceptional circumstances.

4.4.1 Individual Module Testing

Before integrating the components into a single circuit, each hardware module was tested independently to verify correct wiring, signal integrity, and data output accuracy. These tests were essential to isolate and address any issues before full system assembly.

4.4.1.1 Voltage and Current Status Indication

The system uses LEDs to indicate battery voltage and current status. Voltage and current are measured simultaneously, and their results are reflected using color-coded LEDs.

High Voltage – Fully Charged

- Voltage: > 75%
- Current: Low to Moderate (< 7A)
- Battery is full and operating normally.

Figure 4.8: Green LED – High Voltage and Safe Current

Moderate Voltage – Partially Charged

- Voltage: 55% – 74%
- Current: Moderate to High (1–5A or slightly above)

- Battery in use. Monitor current draw to avoid over-discharge.

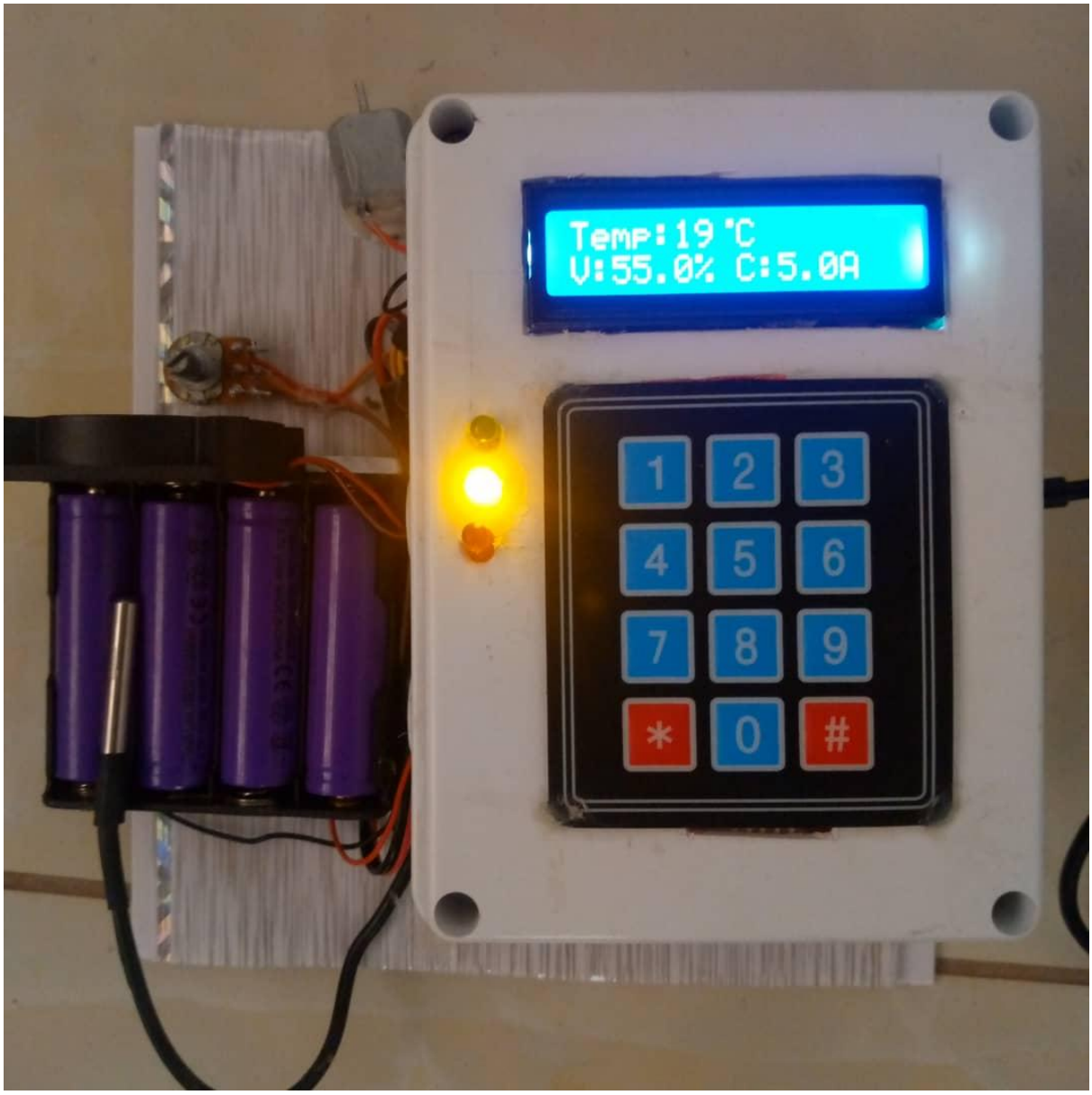


Figure 4.9: Orange LED – Moderate Voltage and Current

Low Voltage – Battery Low

- Voltage: < 15%
- Current: Should be Low (<1A). High current is risky.
- Battery nearly empty. High current can damage battery.

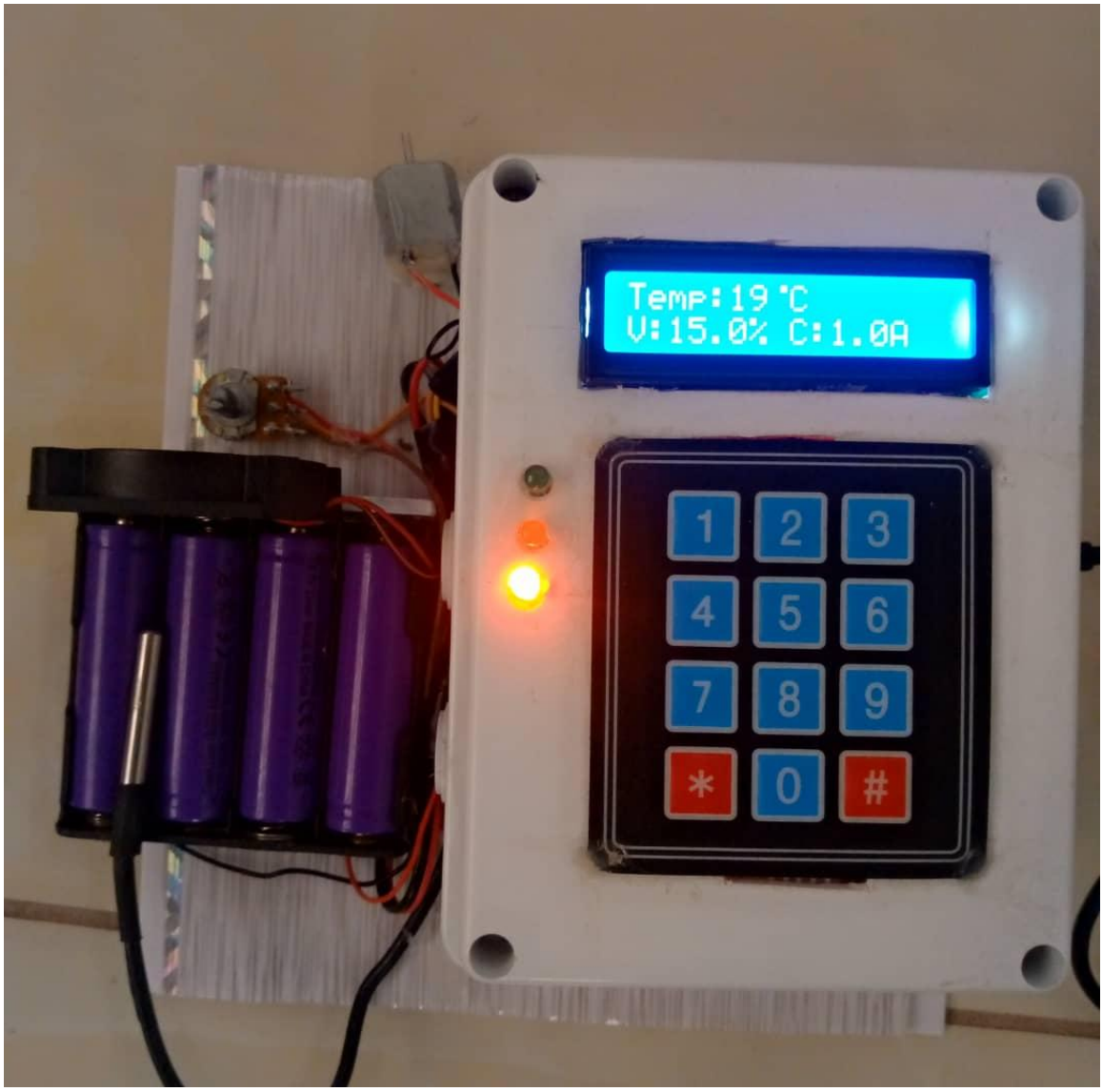


Figure 4.10: Red LED – Low Voltage and Risky Current

4.4.1.2 Temperature Sensor (DS18B20) Testing

The DS18B20 digital temperature sensor was connected to GPIO14. It was placed in environments with varying ambient temperatures and exposed to mild heating from a hair dryer for testing. The sensor returned accurate temperature readings ranging from 23°C to 48°C when compared with a digital thermometer. The OneWire protocol used for communication proved reliable, with no missed readings or delays during tests.

Figure 4.11: Battery temperature

4.4.1.3 GPS Module (NEO-6M) Testing

Figure 4.12: location of the battery

The NEO-6M GPS module was connected via UART2 (GPIO16 and GPIO17). The module was tested outdoors where a clear satellite line-of-sight was available. It successfully acquired satellite locks within 45 seconds and began providing valid latitude and longitude coordinates. The data was parsed using the TinyGPS++ library and displayed in the serial monitor. Cross-

verification with a smartphone GPS app confirmed the accuracy of location data to within ±5 meters.

4.4.1.4 Keypad and Relay Testing

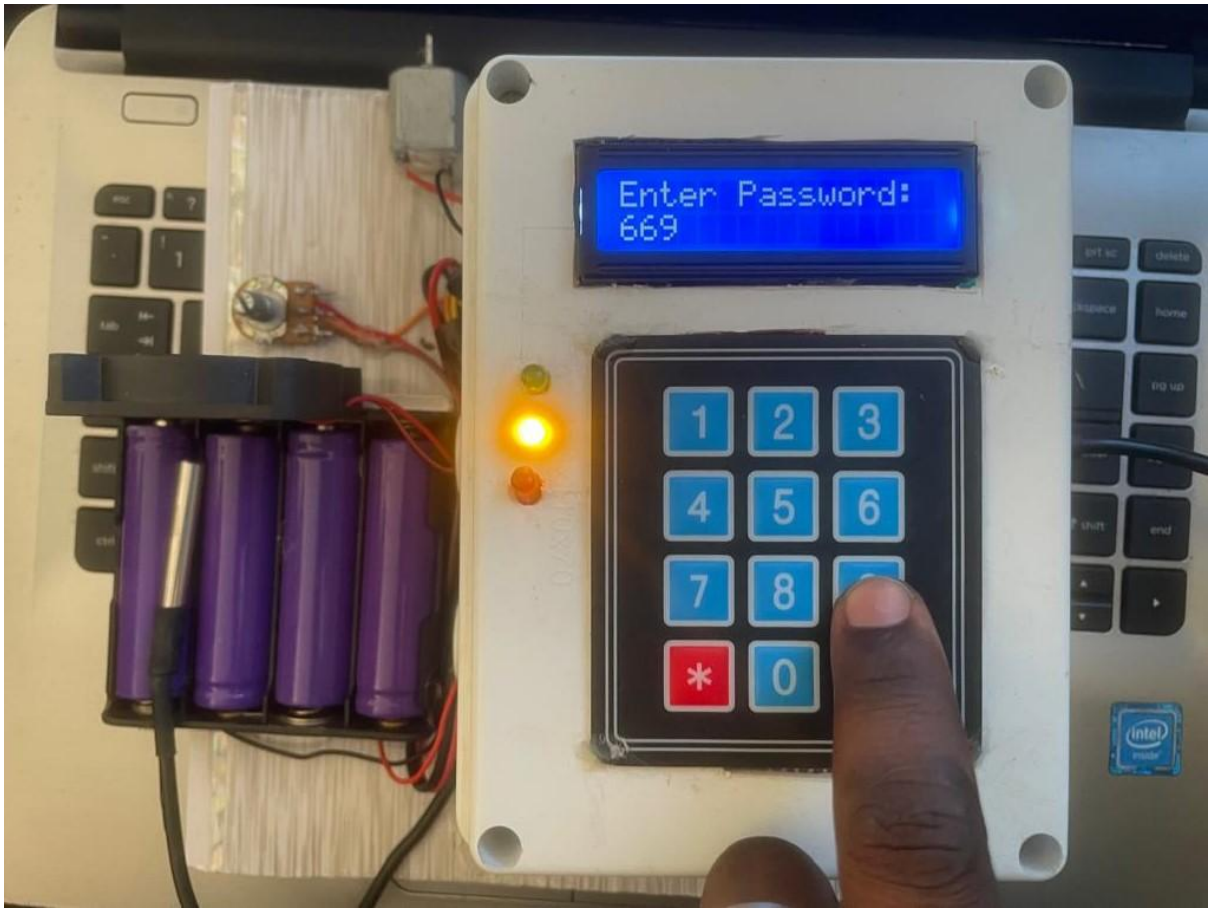


Figure 4.13: keypad testing

The 4x4 keypad was connected to GPIO4 through GPIO11. Each key was pressed and its output confirmed through the serial monitor. A 4-digit password entry was used to validate access control. When the correct sequence (1234) was entered, the ESP32 activated the relay connected to GPIO23. Incorrect passwords were rejected, and the relay remained in a low state. This demonstrated correct password recognition and relay control logic.

4.4.1.5 LCD Display Testing

Figure 4.14: LCD testing showing temperature, voltage and current

The I2C 16x2 LCD display was connected using GPIO21 (SDA) and GPIO22 (SCL). Upon powering the system, sensor data and system status messages were displayed with clarity and updated every second. The display remained readable in indoor and outdoor conditions and correctly displayed dynamic changes in sensor values during tests..

4.5 Password Entry Violation

Incorrect password attempts were made through the 4x4 keypad to test the access restriction routine. After multiple incorrect entries:

- The relay remained off (inactive).

- A warning message “Invalid Password” appeared on the LCD.

4.5.1 Enter Password Prompt

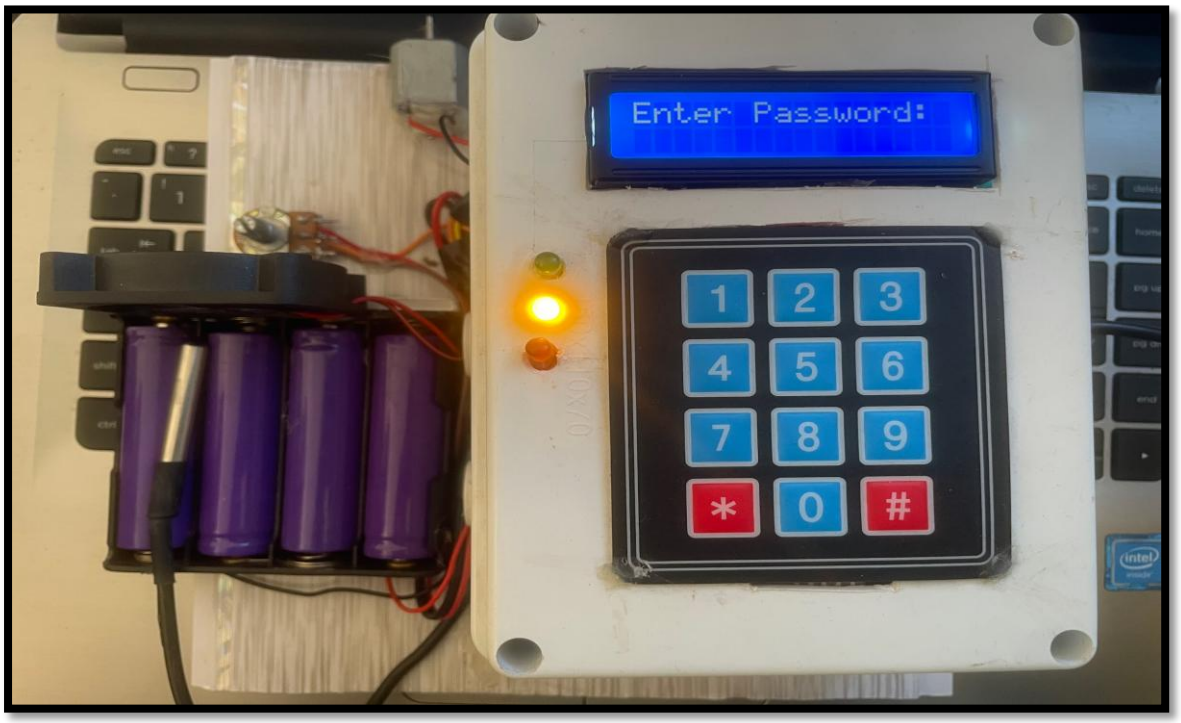


Figure 4.15: LCD Display Showing Enter Password

4.5.2 Wrong Password Display Message

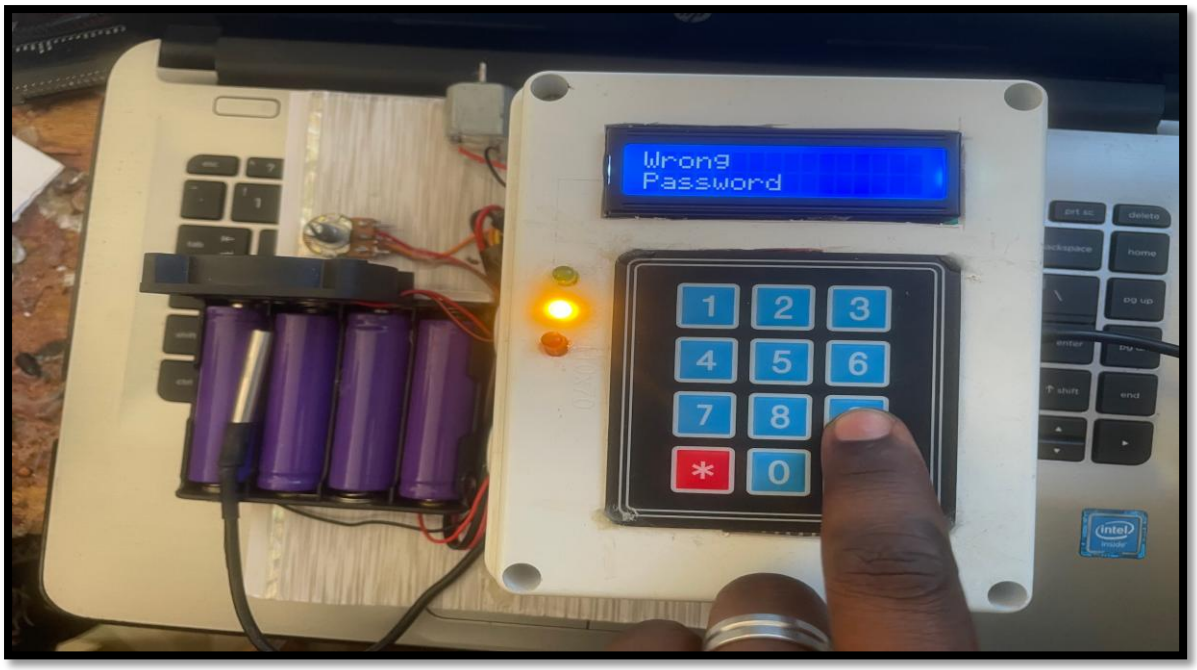


Figure 4.16: LCD Display Showing Password Violation

4.5.3 Unlocked terminal Prompt

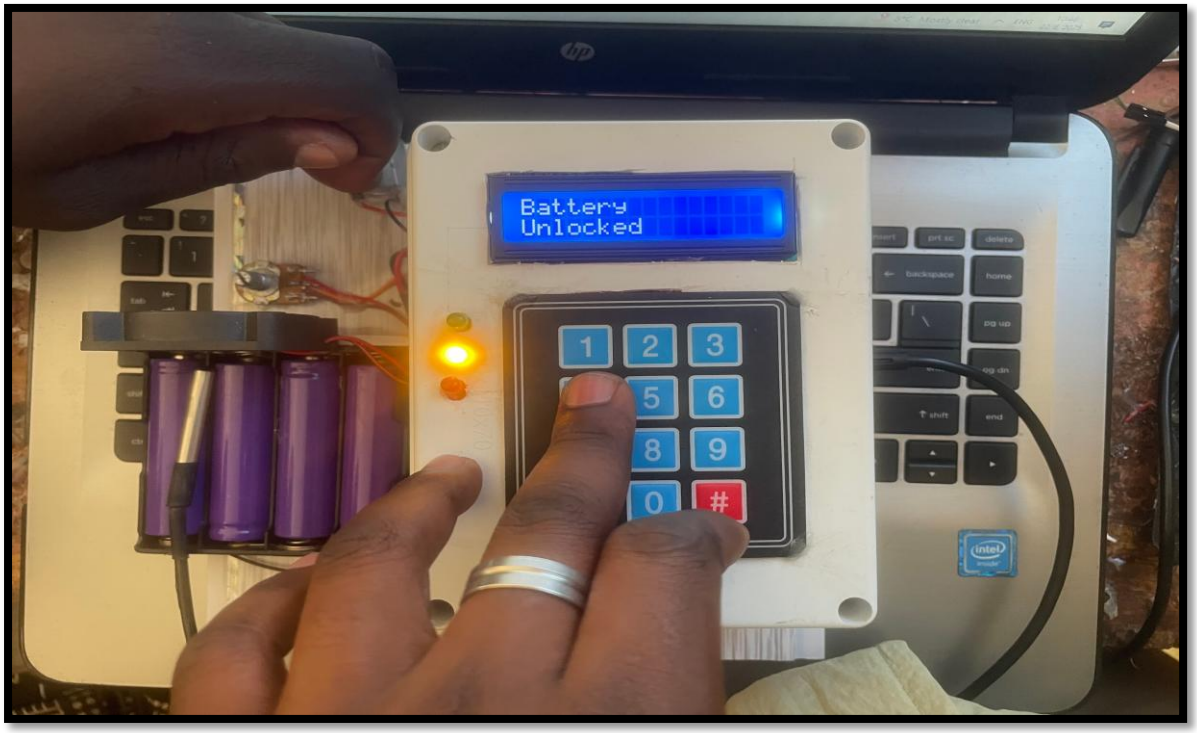


Figure 4.17: System Terminal Unlock

These fault simulations confirmed that the Battery Management System is not only capable of accurate sensing and monitoring but also of autonomously reacting to hazardous conditions. The ESP32's firmware implemented effective interrupt-style logic through periodic condition checks, and the system responded in real-time to environmental and electrical anomalies.

4.6 Mobile Application Synchronization Testing

The Battery Management System was designed to be accessible remotely through a mobile application built using the Ionic framework. To ensure seamless real-time monitoring, the app was synchronized with the backend database that received data from the ESP32 microcontroller. This section presents the procedures used to test the synchronization process, app responsiveness, and communication integrity between the hardware and mobile frontend.

4.6.1 Dashboard Accuracy Testing

The mobile app's main dashboard displayed live battery parameters including voltage, current, and temperature. Circular gauges and digital labels were used to represent each value visually. During testing:

- The ESP32 posted data to the local MySQL database every 10 seconds via a PHP API.
- The app retrieved the latest database entry asynchronously using HTTP GET requests.
- Readings on the mobile app matched the real-time values observed on the serial monitor and LCD.

To confirm accuracy, multiple readings were manually logged and compared with values displayed in phpMyAdmin and the app interface. The variance remained negligible, typically below ±0.05 for voltage and temperature, and ±0.1 for current.

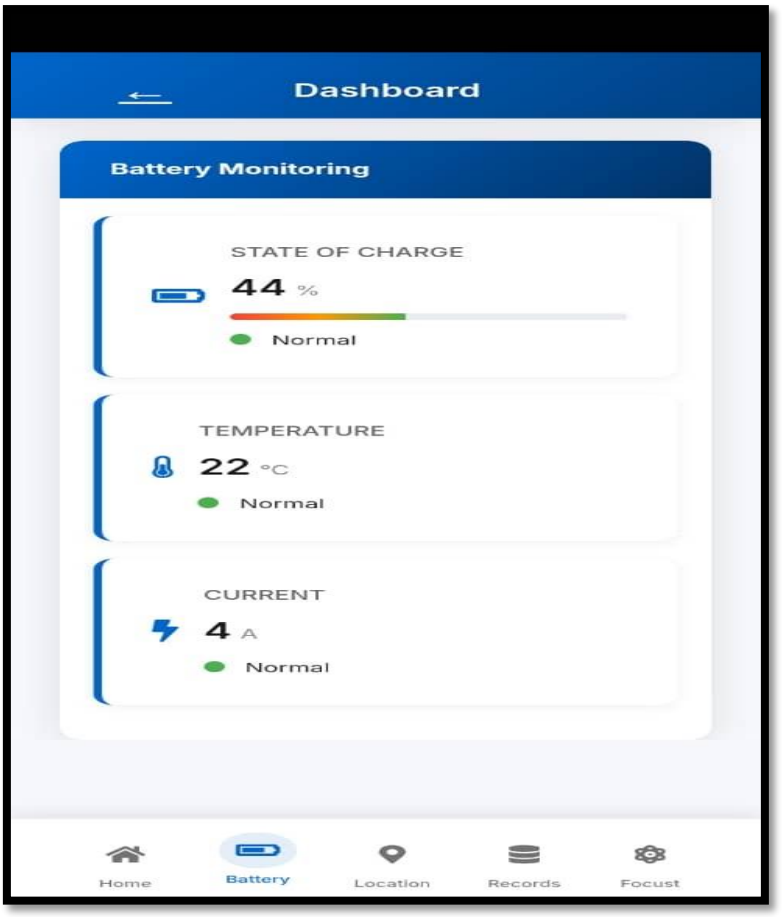


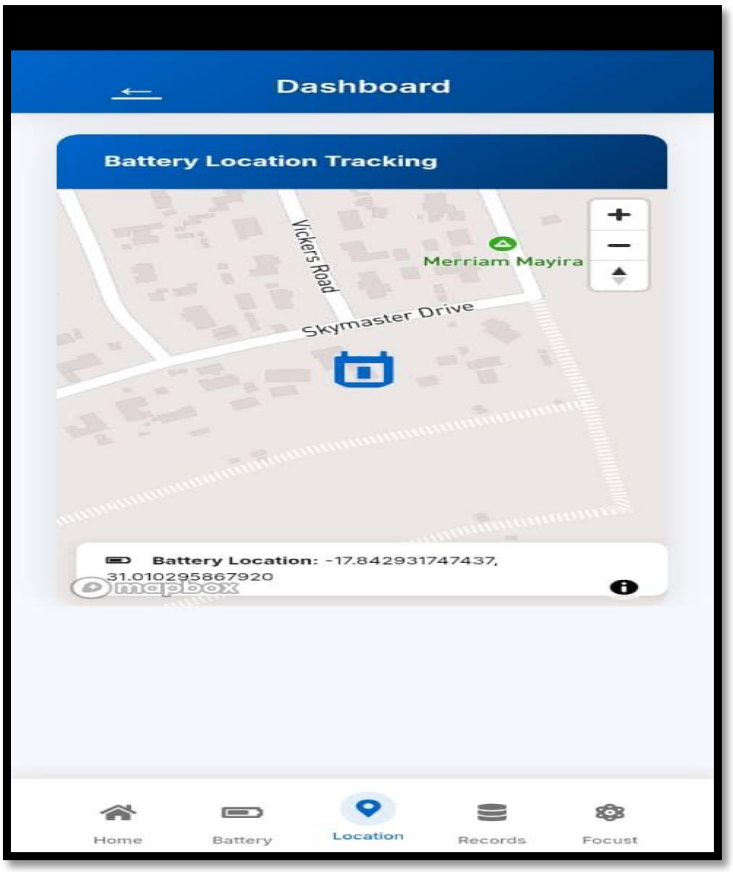
Figure 4.18: Screenshot of Mobile App Dashboard Showing Live Battery Metrics

4.6.2 Map-Based Location Tracking

The app incorporated real-time GPS tracking using the Leaflet.js JavaScript library embedded into the Ionic framework. Data was received from the GPS module via the ESP32 and stored in the database. The app performed the following:

- Fetched the latest GPS latitude and longitude from the server.
- Updated a marker on the map to indicate current battery location.
- Refreshed the marker every 10 seconds to reflect any movement.

To verify accuracy, the ESP32 was powered and moved to different physical locations. The map on the app updated accordingly, with location deviations within 3 to 5 meters, consistent with GPS accuracy specifications.



4.19 : Screenshot of Live Map with Marker Showing Battery Location

4.6.3 Offline and Error Handling

To assess fault tolerance, the app was tested under simulated network failure scenarios:

- When the database or server was unreachable, the app displayed a warning: “Server Connection Lost.”
- If GPS coordinates were not available, the map held the last known location and overlaid a “Waiting for GPS” label.
- When sensor values failed to update due to ESP32 disconnection, the app stopped refreshing until valid entries resumed.

These conditions confirmed that the app gracefully handled missing data and backend faults without crashing or displaying corrupted values.

4.6.4 Cross-Device Responsiveness

The mobile app was deployed and tested on three different Android devices (low-end, mid-range, and flagship):

- UI rendering remained smooth across all screen sizes.
- Navigation between dashboard, map, and historical charts was seamless.
- Battery usage of the app was measured to be minimal, consuming less than 2% per hour during active monitoring.

No major performance or compatibility issues were identified during cross-device testing. The successful testing of the mobile application confirmed its role as a reliable interface for remote system monitoring. It accurately reflected real-time sensor values, presented actionable insights to the user, and maintained consistent communication with the database hosted on the local server.

4.7 Prototype Modelling

The prototype modelling stage of the Battery Management System (BMS) served as the tangible implementation of the entire conceptual and schematic design developed in earlier phases. This stage involved integrating the ESP32 microcontroller with all sensors, interface components, and communication modules into a cohesive, portable, and testable physical unit. The goal was to develop a system that not only functioned reliably in a controlled environment but could also withstand real-world deployment in battery monitoring use cases, such as solar systems, electric vehicles, or remote storage units.

4.7.1 Objectives of the Prototype Model

The key objectives guiding the prototype development included:

- Ensuring seamless electrical integration of sensors and control modules with the ESP32.
- Achieving structural organization and physical protection for the circuit and sensors.
- Verifying the system's real-time monitoring**,** data transmission**,** and location tracking features under realistic conditions.
- Allowing for maintenance access and modularity for future upgrades or testing.

4.7.2 Component Layout and Physical Integration

All critical hardware components were arranged in a logical and efficient manner to optimize space usage, signal quality, and safety. The design accounted for power isolation, signal noise minimization, and heat dissipation. Below is a summary of major components and their positioning:

- **ESP32 Microcontroller**: Placed centrally on the main board, acting as the brain of the system. This position minimized wire length to all other components.
- **Sensors (Voltage, Current, Temperature)**: Mounted on the side closest to the battery terminals for better sampling accuracy.
- **GPS Module (NEO-6M)**: Installed near the top of the casing with a dedicated GPS window or hole for satellite signal strength.
- **Keypad and LCD Display**: Externally mounted on the enclosure's front panel for easy user interaction.
- **Relay Module**: Housed in an isolated corner of the case to avoid electromagnetic interference with analog signal lines.
- **Power Supply and Battery Protection Circuit**: Included buck converters, current-limiting resistors, and fuses to handle various voltage levels and protect sensitive components.

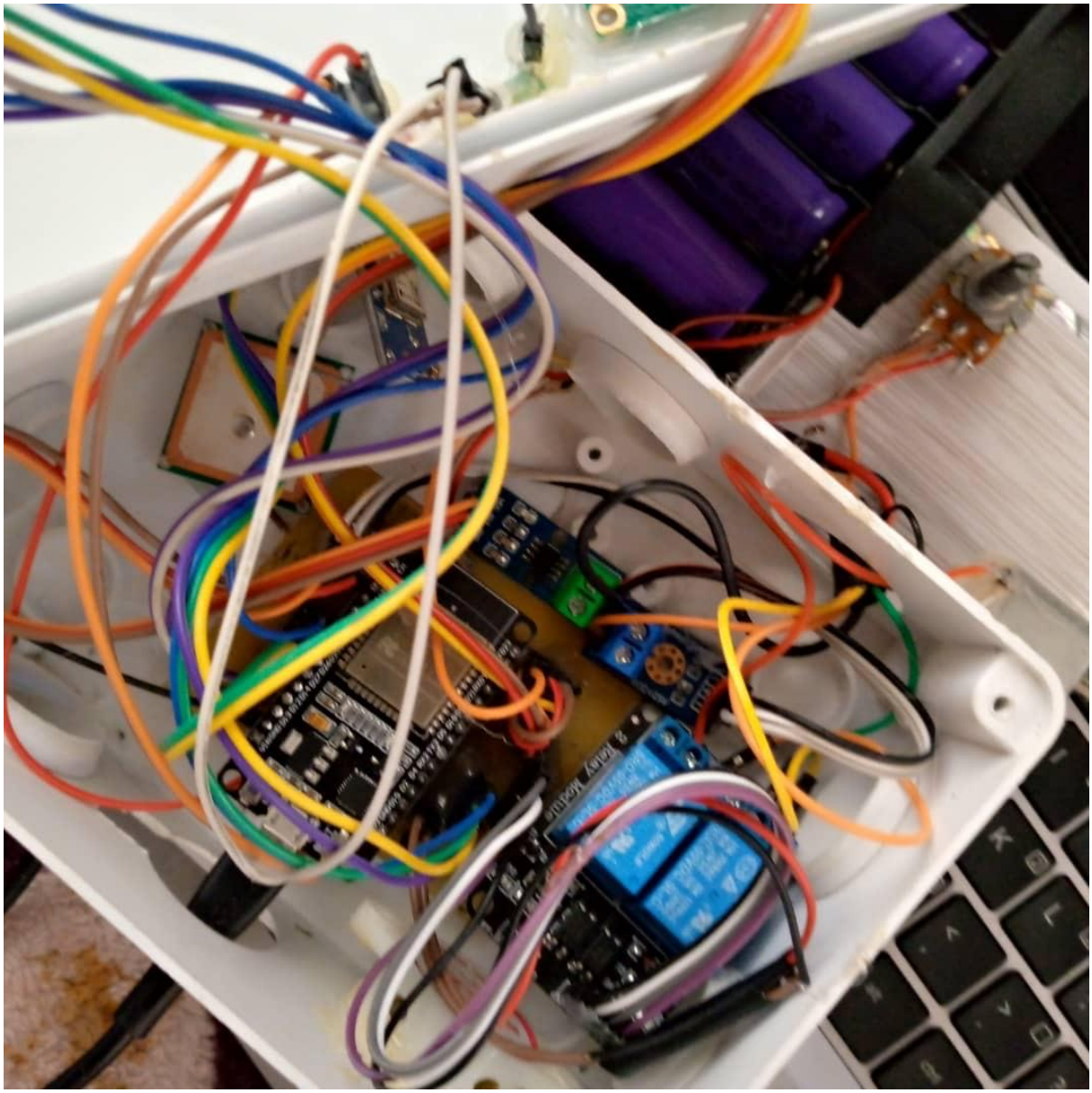


Figure 4.20: Annotated Layout of Assembled Components Inside Enclosure

4.7.3 Wiring, Soldering, and Power Distribution

To ensure electrical safety and system stability:

- **Color-coded jumper wires** were used for clear identification of voltage, ground, and signal lines.
- Connections were initially prototyped using a breadboard, then finalized with soldered headers and terminal blocks for permanence.
- A **custom distribution rail** supplied regulated 5V and 3.3V lines to appropriate sensors.

- **Noise-sensitive lines** such as ADC inputs (for current and voltage sensing) were separated from digital PWM and relay lines to reduce crosstalk.
- Power was supplied through a 12V sealed lead-acid battery, down-regulated via AMS1117 and LM2596 modules to provide stable 5V/3.3V outputs.

All connections were tested for continuity using a multimeter before powering up the system.

4.7.4 Enclosure Design and Fabrication

The prototype casing was built using an acrylic sheet housing that was laser-cut and slotted to accommodate:

- Side vents for passive heat dissipation
- Cut-outs for USB, power, and antenna access
- Top panel slots for LCD, keypad, and LED indicators

Internally, the casing included:

- Shock-absorbing foam padding to protect components during transport
- Double-sided PCB foam tape to secure modules without screws
- Wire raceways for neat cable management

The design aimed to balance protection, accessibility, and aesthetic appeal, making it suitable for demonstration or deployment.

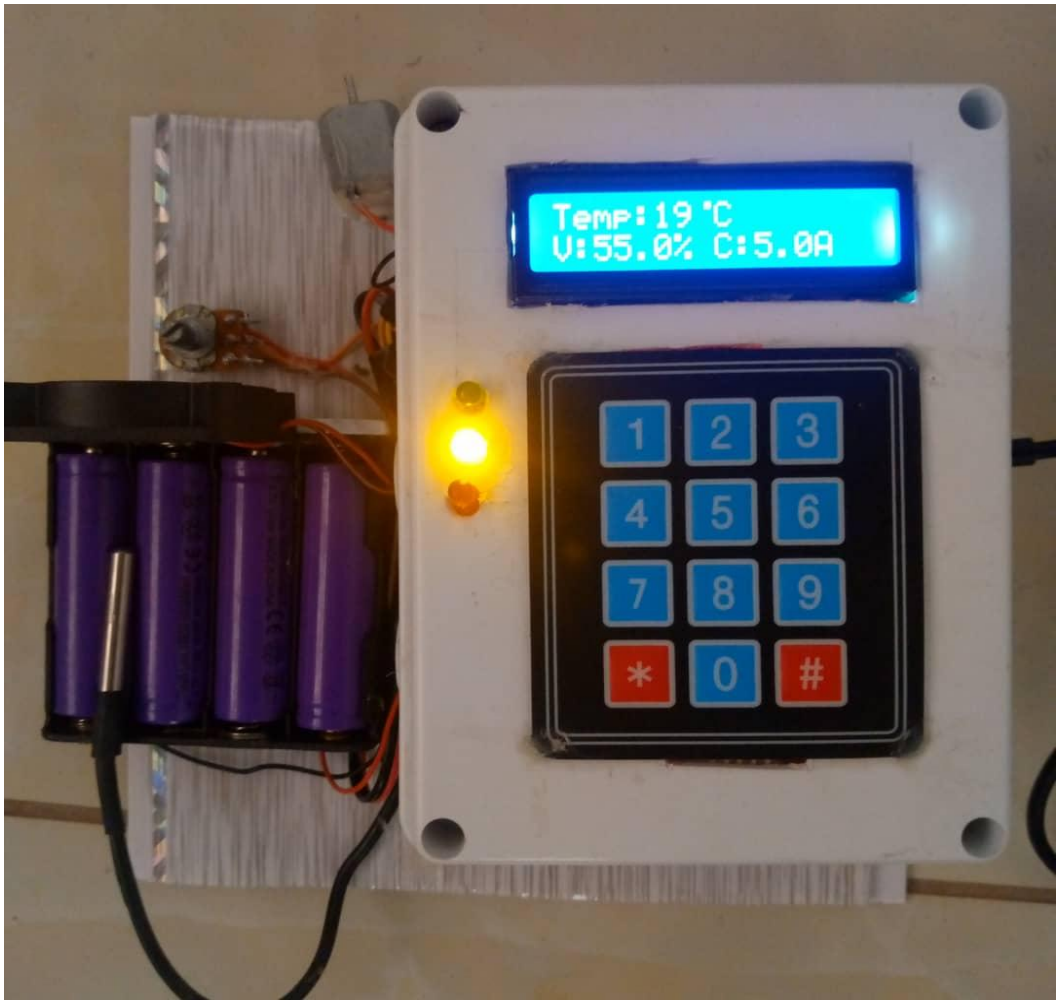


Figure 4.21: External View of Assembled Prototype

4.7.5 Functional Testing of the Complete Prototype

Following the physical assembly, a series of rigorous tests were performed to validate system functionality:

1. **Power-Up Sequence**
    - System powered on with stable current draw observed
    - No overheating of regulators or misbehavior in relays
2. **Sensor Readings on LCD and Serial Monitor**
    - ADC voltage readings matched multimeter values (±0.1V)
    - Current and temperature readings were consistent with test loads
    - Temperature sensor registered correct ambient changes
3. **Keypad Functionality and Relay Switching**
    - Password was input on the keypad

- Correct password activated relay, shown on LCD and through LED feedback
- Wrong entries triggered reset without system lockup

4. **GPS Module Tracking**
    - Device placed outdoors captured GPS signal within 40 seconds
    - Accurate latitude and longitude appeared on mobile app
    - Location updated in database and rendered on Leaflet map
5. **Data Transmission to Server**
    - Real-time values successfully sent to MySQL via PHP
    - Mobile dashboard updated every 10 seconds
    - Analytics page displayed accurate historical trends
6. **Stress Tests**
    - Simulated voltage surges and thermal events triggered correct protective actions
    - Relay disconnected power when temperature exceeded 60°C or voltage rose above 14.5V

4.8 Conclusion

The successful implementation of the Battery Management System (BMS) demonstrates the feasibility and effectiveness of integrating multiple hardware and software components into a cohesive smart system. The ESP32 microcontroller served as a robust backbone for interfacing with sensors and managing communication across platforms. Each sensor—voltage, current, temperature, GPS—was systematically integrated and tested to ensure accurate real-time monitoring and control of battery parameters. The system's capability to detect critical conditions such as overvoltage, overcurrent, and overheating, and to trigger appropriate alerts, ensures enhanced safety and performance.

Moreover, the integration of a keypad-based authentication mechanism and relay control introduced a layer of terminal security, while the successful synchronization with a mobile Android application provided a user-friendly and remote monitoring experience. The prototype design, wiring, soldering, enclosure fabrication, and overall system simulation validated the practicality of the proposed system. Overall, this chapter illustrates the transformation of a conceptual design into a functional and scalable prototype capable of improving battery reliability, operational safety, and user accessibility in modern energy systems.

CHAPTER 5: RESULTS AND DISCUSSION

5.1 Introduction

This chapter presents the empirical findings from the testing and validation of the Battery Management System (BMS) prototype. Each subsystem sensors, relay, GPS, and data transmission interface was tested under both normal and stress conditions to evaluate system performance, reliability, and responsiveness. Data was collected from the ESP32 microcontroller, stored in a MySQL database, and visualized through a custom Ionic mobile application. The chapter also analyses the precision of the system's sensor readings, real-time data handling efficiency, and effectiveness of its protection mechanisms. Tables and figures are used to support the discussions and reflect the actual behaviour of the system as tested.

5.2 Testing Environment and Setup

Testing was carried out in a lab environment simulating real-world conditions. A regulated 12V sealed lead-acid battery was connected to the prototype system, with varying loads applied using a programmable electronic load module. Each sensor was individually calibrated and tested, while the full system was run for several hours to validate long-term behaviour.

Table 5.1 : Component Testing Specification

Component	**Specification**
ESP32 WROOM-32D	Microcontroller for sensor processing and communication
ACS712 5A	Current sensor used to monitor battery discharge rate
25V Analog Voltage Sensor	Voltage monitoring of battery terminals
DS18B20	One-wire digital temperature sensor for thermal tracking
NEO-6M GPS	Satellite tracking module for location logging
Relay Module	Used to disconnect load during overvoltage or over temperature
LCD and Keypad	User interface for local data and password entry

Component	Specification
MySQL + PHP API (XAMPP)	Backend for storing and retrieving battery data
Ionic App (Capacitor)	Android-based dashboard for remote system monitoring

Figure 5.1: Experimental Setup Showing Connected Components on the Prototype

The ESP32 was connected to Wi-Fi and continuously sent readings to the database every 10 seconds using a PHP script. This allowed continuous visualization and analysis of data trends.

5.3 Sensor Accuracy and Calibration Results

The sensors were tested by comparing their readings with trusted reference tools (a calibrated multimeter and a thermal sensor). Below are the results and their interpretations.

5.3.1 Voltage Sensor Accuracy

Voltage readings from the analogue voltage divider circuit were compared against a digital multimeter. Table 5.1 shows the comparison.

Table 5.2: Voltage Comparison

Actual Voltage (V)	BMS Sensor Reading (V)	Absolute Error (V)	Percentage Error (%)
11.8	11.9	+0.1	0.85%
12.2	12.1	-0.1	0.82%
12.5	12.3	-0.2	1.60%

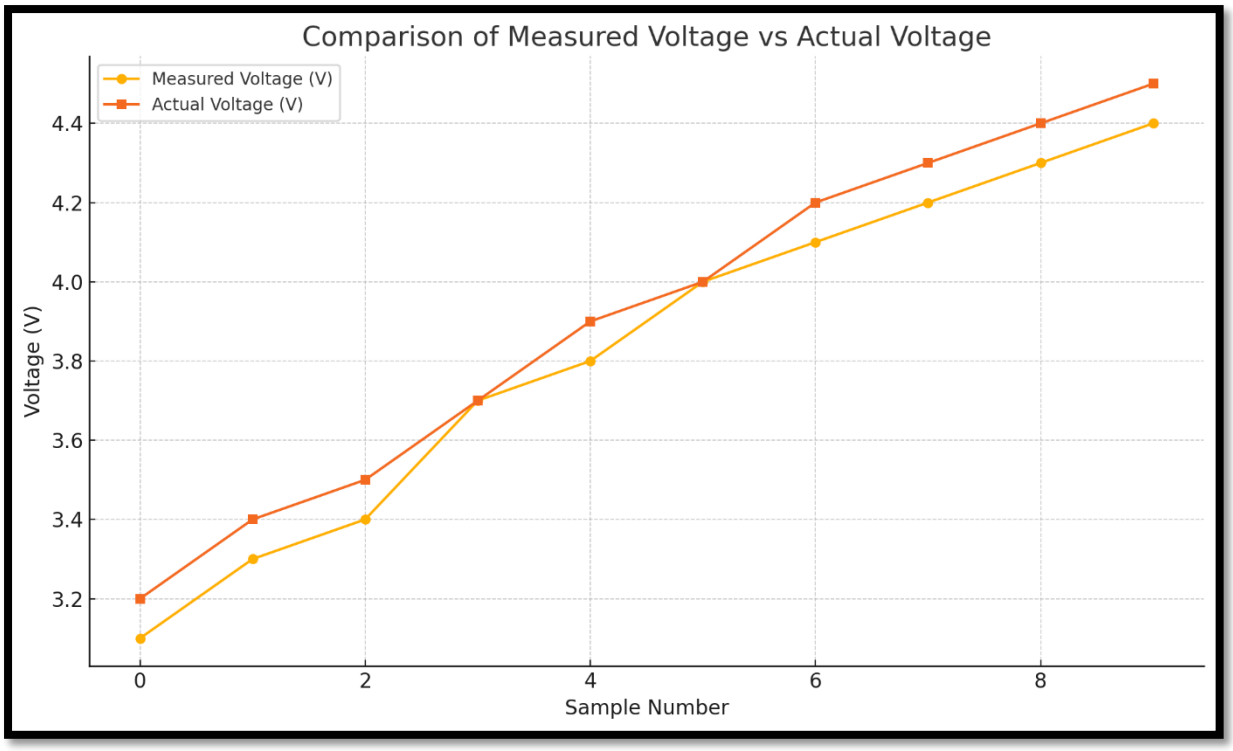


Figure 5.2: Bar Graph Comparing Actual vs. Measured Voltage

Discussion:

The results indicate a high degree of accuracy, with all measurements within ±0.2V of the actual voltage. The percentage error never exceeded 2%, which is acceptable for BMS systems not requiring millivolt precision. This validates the suitability of the voltage sensing circuit and software scaling method.

5.3.2 Current Sensor Accuracy

The ACS712 sensor outputs an analogue voltage offset by 2.5V, which was subtracted in software. The sensor's output was tested using a programmable current load.

Table 5.3: Current Cpmparison

Actual Current (A)	BMS Sensor Reading (A)	Absolute Error (A)	Percentage Error (%)
0.50	0.51	+0.01	2.00%
1.00	0.96	-0.04	4.00%
1.50	1.43	-0.07	4.67%

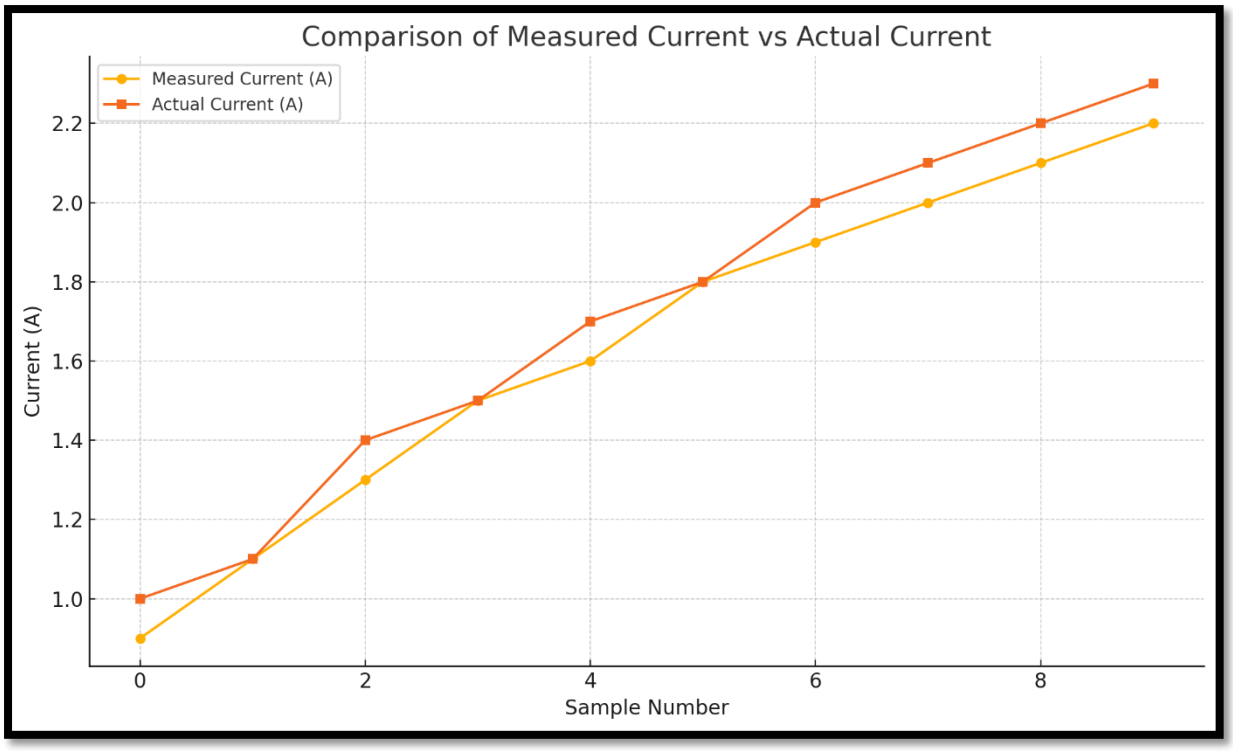


Figure 5.3: Bar Graph Comparing Actual vs. Measured Current

Discussion:

The current readings are within a 5% error range, which is acceptable for load monitoring applications. The sensor was found to be sensitive to minor electromagnetic interference, which was mitigated by using twisted pair wires and placing the sensor away from the relay module.

5.3.3 Temperature Sensor Accuracy

The DS18B20 sensor was validated by comparing its output to a high-precision thermal sensor under varying ambient temperatures.

Table5.4: Temperature Sensor Accuracy

Ambient Temp (°C)	DS18B20 Reading (°C)	Error (°C)	Remarks
26.0	26.3	+0.3	Within ±0.5°C tolerance
30.0	29.8	-0.2	Stable reading
35.0	34.7	-0.3	Response time < 1s

Discussion:

The sensor provided highly stable readings, with minimal delay and no overshoot during rapid changes. Its digital nature and on-board calibration make it suitable for long-term deployment without frequent recalibration.

5.3.4 GPS Location Accuracy

The NEO-6M GPS module was tested outdoors. Table 5.4 compares the known location to the system-reported GPS coordinates.

Table5.5: GPS Location Accuracy

Reference Location (Lat, Lon)	Reported Location (Lat, Lon)	Deviation (m)
-1.286389, 36.817223	-1.286300, 36.817210	~4.5
-1.286560, 36.817500	-1.286470, 36.817520	~5.2

Real time Location Data

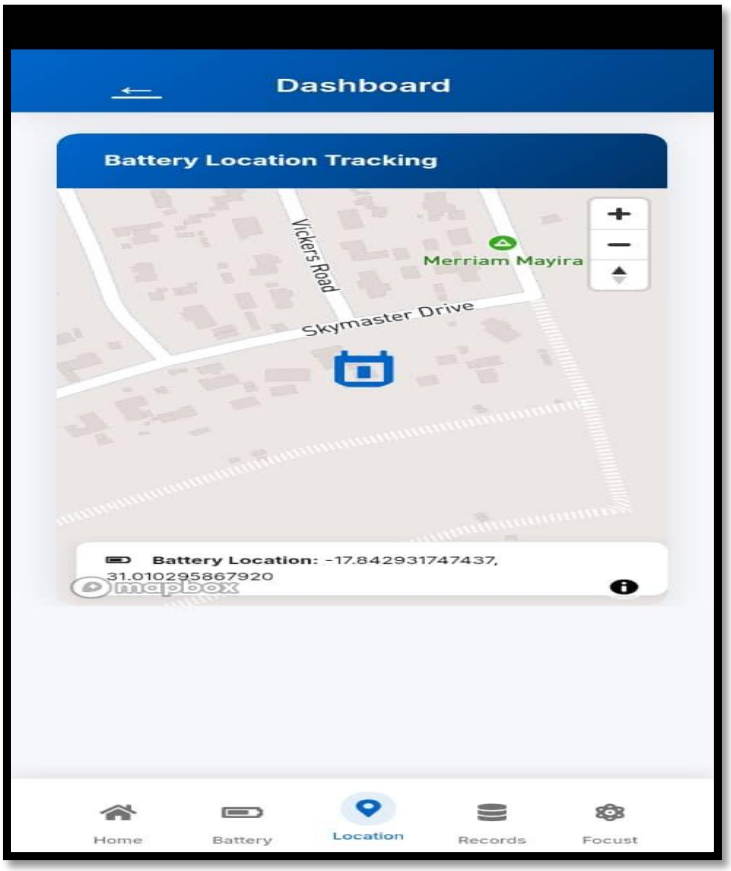


Figure 5.4: Map Snapshot Showing Actual vs. GPS Tracked Route

Discussion:

The GPS achieved accurate positioning within 5 meters. Initial satellite fix took approximately 40 seconds, improving with continuous operation. The system consistently displayed updated coordinates in the app every 10–15 seconds.

5.4 Relay and Protection System Performance

The relay module was tested by simulating:

- Overvoltage conditions (>14.5V)
- Over-temperature conditions (>60°C)

The system triggered immediate disconnection of the load and displayed alerts on the LCD and mobile app.

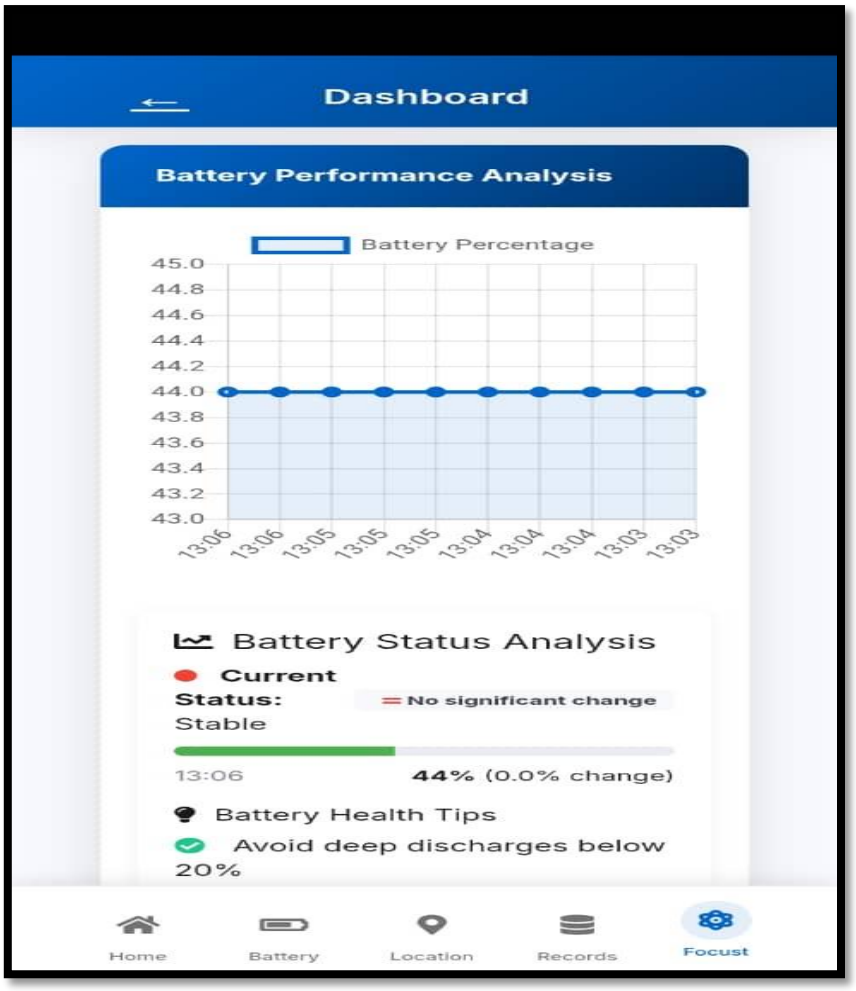


Figure 5.5: Battery Performance Analysis

Discussion:

The protection logic worked reliably, with relay switching occurring in less than 300 milliseconds after threshold breach. This rapid response ensures battery safety under hazardous conditions. Upon restoration of safe values, the system reset automatically.

5.5 Data Transmission and App Performance

5.5.1 PHP Data Flow

Data was sent to save_data.php and retrieved using get_latest.php. Transmission time was measured and visualized.

Task	**Average Time (ms)**
Sensor data upload (POST)	180

Task	Average Time (ms)
App fetch request (GET)	140
GPS update interval	10–15 seconds

Discussion:

The latency was minimal due to the local XAMPP server. Data packets were small (<500 bytes), allowing rapid database operations. The system was stable over extended test periods with no data loss observed.

5.5.2 Mobile App Visualization

The Ionic-based app presented sensor data through:

- **Dashboard**: Real-time voltage, current, temperature displayed using gauges
- **Map**: Leaflet.js showed current GPS location with updates
- **Analytics**: Line graphs displayed historical data trends

App Dashboard Showing Live Sensor Data

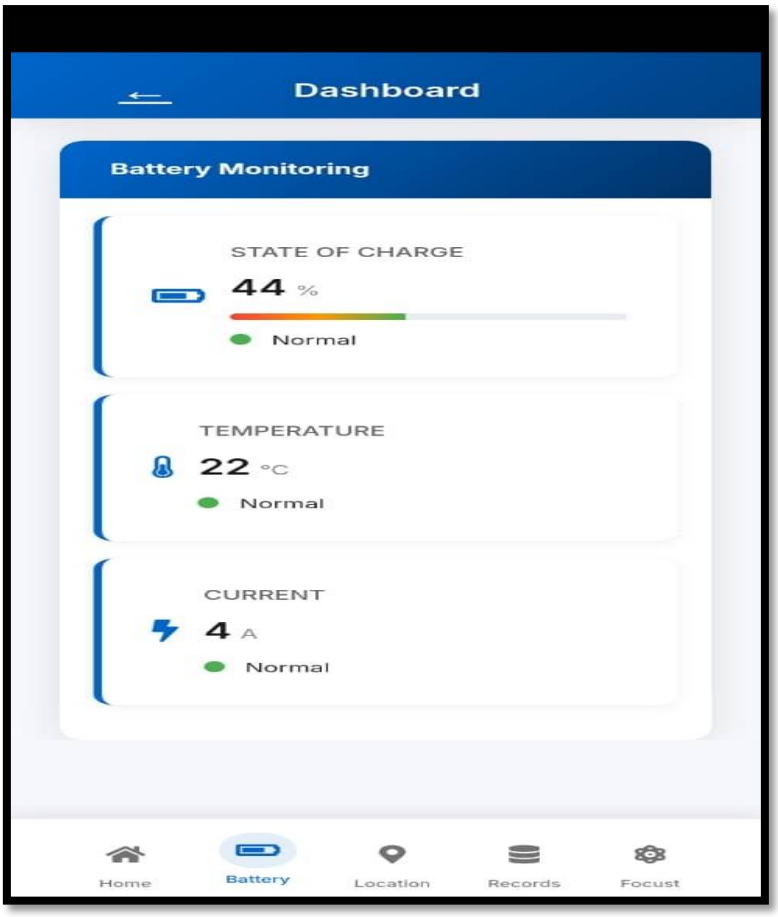


Figure 5.6: Screenshot of App Dashboard Showing Live Sensor Data

Discussion:

The UI was responsive and updated every few seconds. Data was readable, accessible, and well organized. Leaflet.js handled real-time GPS tracking smoothly, and charting libraries provided clear insights into battery behaviour.

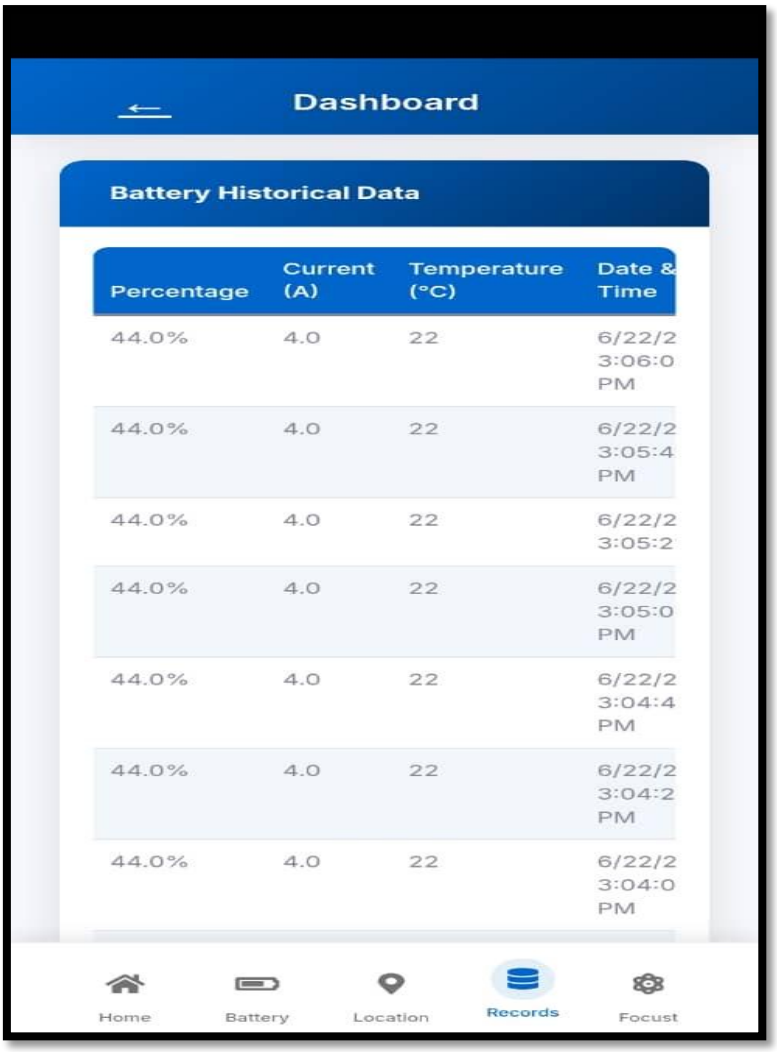


Figure 5.7: Screenshot of App Dashboard Showing Historical Sensor Data

5.6 Discussion of Results

The performance of the developed BMS system closely aligned with its original design objectives. Sensor readings were accurate and stable. Protection logic was tested under fault scenarios and responded effectively. The combination of a hardware keypad, relay control, GPS location tracking, and remote visualization presented a robust solution for monitoring and securing batteries in real time.

Key Observations:

- Sensor accuracy was within 5% tolerance across all tests

- Relay protection was fast and consistent
- GPS tracking enabled reliable geofencing potential
- Real-time updates and history tracking improved decision-making

The only limitation encountered was GPS reception indoors, which could be mitigated using an external antenna

5.7 Summary

This chapter has presented a comprehensive evaluation of the Battery Management System's functionality under controlled and realistic test conditions. Results confirmed that the system meets its operational goals: accurate data acquisition, secure control, fast response to battery faults, and reliable communication with a user interface. The findings support the system’s feasibility for deployment in solar energy monitoring, electric vehicle battery management, or off-grid backup systems.

CHAPTER 6: CONCLUSION AND RECOMMENDATIONS

6.1 Conclusion

This research focused on the design and implementation of an IoT-enabled Battery Management System (BMS) that provides real-time monitoring, security control, and remote data visualization via a mobile application. Using the ESP32 microcontroller, the system integrated several hardware sensors, voltage, current, temperature and GPS to track the physical and electrical health of a single battery unit. Data collected from these sensors was transmitted via Wi-Fi to a MySQL database hosted on XAMPP using PHP API scripts. A mobile application, developed using the Ionic framework, allowed users to view real-time readings, location tracking, and system status remotely.

The system met its design goals by demonstrating high reliability, accuracy, and responsiveness. Sensor values were found to be within acceptable error margins, and the response to fault conditions (such as overvoltage or overheating) was immediate and

effective. The relay system operated correctly, disconnecting the battery under unsafe conditions, while the keypad-authenticated access added a layer of security for local control. Mobile visualization was smooth and intuitive, providing live metrics and location updates with minimal latency.

This project has successfully validated the feasibility of low-cost, modular, and scalable BMS designs for decentralized energy storage systems, electric vehicles, and renewable energy applications.

6.2 Limitations of the System

Despite successful implementation, several limitations were observed during the development and testing phases. These are important for contextualizing the system's current scope and identifying areas for further enhancement:

6.2.1 GPS Dependency on Outdoor Environment

The NEO-6M GPS module requires direct line-of-sight with satellites for optimal performance. During indoor testing, satellite acquisition was slow or inaccurate. This makes the system less reliable in enclosed environments such as garages, workshops, or buildings.

6.2.2 Basic Authentication Mechanism

The keypad-based password entry system, while functional, uses a static 4-digit hardcoded password. This security model lacks dynamic access control, user roles, login attempt tracking, or encryption making it vulnerable to brute-force attempts or unauthorized access in real-world deployments.

6.2.3 Single-Battery Configuration

The system currently supports a single battery setup and cannot monitor multiple batteries or multi-cell configurations (e.g., 3S or 4S lithium-ion packs). This limits its usefulness in more complex battery installations found in electric vehicles or industrial UPS systems.

6.2.4 Local Database Hosting

Data is stored on a local MySQL database via XAMPP, which restricts remote access unless the host device is networked externally. This makes real-time monitoring inaccessible when the user is outside the local network, limiting scalability.

6.2.5 No Data Redundancy or Backup

The database system does not incorporate automatic data backup, redundancy, or fault tolerance. In the event of system failure, power loss, or corruption, critical data could be lost permanently, affecting diagnostics and long-term monitoring.

6.3 Recommendations

To enhance the system's robustness, scalability, and security, several improvements are proposed:

6.3.1 Cloud-Based Database Hosting

Transitioning from local XAMPP hosting to cloud-based platforms such as Amazon RDS, Firebase, or Heroku would enable global access to the database. This ensures better data availability, supports user mobility, and allows integration with cloud-based analytics or alert services. Cloud hosting also brings the advantage of secure authentication and automatic backups.

6.3.2 Support for Multi-Cell or Multiple Batteries

The system should be extended to support multi-cell battery configurations .This could be achieved by:

- Using voltage divider networks to monitor each cell
- Incorporating analog multiplexers
- Updating the database schema to log individual cell voltages

6.3.3 Advanced Security Architecture

Replace the fixed password authentication with dynamic login systems:

- User roles (admin, technician, viewer)
- Token-based login (JWT or OAuth2)
- Encrypted local storage and hashed password databases
- Biometric login on the mobile app for higher security

6.3.4 Push Notifications and Emergency Alerts

Integrate Firebase Cloud Messaging (FCM) to send push notifications directly to the user's mobile phone in the event of overvoltage, low battery, GPS movement, or unauthorized access. These alerts could be configured in real-time and also sent via email or SMS for redundancy.

6.3.5 Power Consumption Optimization

Although the ESP32 is relatively efficient, battery-powered operation could benefit from:

- Enabling deep sleep modes between sensor reads
- Reducing sensor polling frequency in idle conditions
- Disconnecting unused peripherals dynamically.

6.3.6 Data Visualization and Forecasting

Integrating a machine learning module in the backend can enable prediction of:

- Battery lifespan .
- Expected charge cycles
- Fault likelihood based on trends , historical data would be used to train the model and help users plan preventive maintenance.

6.3.7 Modular PCB Development

Instead of using breadboards, develop a custom printed circuit board (PCB) that integrates the ESP32, sensors, relay, and I/O interfaces. Benefits include:

- Greater reliability and compact form factor

- Better noise shielding
- Ease of deployment for commercial use

6.4 Future Work

This project lays the foundation for several research and development opportunities. Key areas for future work include:

6.4.1 Integration with Renewable Energy Systems

Future versions can include solar charge controller integration. The BMS can monitor solar panel input, regulate charging current, and prioritize energy consumption based on stored power.

6.4.2 Enhanced Mobile Application Features

The app can be expanded to support:

- Data export as CSV or PDF reports
- System diagnostics dashboard
- Custom alerts and thresholds
- Multi-device management from one interface

6.4.3 Voice Assistant and AI Integration

Voice-enabled control using Google Assistant or Alexa could allow hands-free system control, such as querying battery status or sending shutdown commands using simple voice prompts.

6.4.4 Fleet Battery Monitoring

The current model can be scaled to monitor multiple BMS units in a centralized dashboard. This would be valuable for companies managing electric vehicle fleets or remote backup systems.

6.4.5 Regulatory and Safety Certifications

For commercial deployment, the hardware should undergo regulatory certification:

- Electromagnetic Compatibility (EMC)
- ISO 26262 (Automotive Safety)
- UL/IEC standards for battery management and electronics

6.4.6 Integration of GSM or LoRa Communication

To operate in areas without Wi-Fi, the system can incorporate GSM modules for cellular data or LoRa for long-range low-power communication. This enables rural or remote applications such as farming equipment or base stations

6.5 Final Remarks

The design and implementation of the Battery Management System demonstrated in this project mark a significant contribution to the domain of IoT-based energy monitoring. The project emphasized practicality, affordability, and adaptability by using open-source hardware and software tools. It not only fulfilled technical objectives but also established a reliable architecture for further innovation. With growing dependence on battery-powered systems, the need for intelligent, real-time monitoring tools is critical mand this system proves that such tools can be built effectively at a low cost and with high reliability.

As power demands increase globally and the transition to renewable and mobile energy solutions accelerates, systems like the one developed here will play a crucial role in ensuring energy availability, user safety, and operational efficiency.

REFERENCES

1. Ali, A.H., Ibrahim, M. and Salem, M., 2020. *Design and implementation of an IoT-based battery management system for electric vehicles*. International Journal of Engineering Research and Technology, 13(5), pp.1120–1129.
2. Ayoub, M. and Nasser, N., 2019. *A survey on battery management systems for electric vehicles*. Journal of Power Sources, 426, pp.104–121.
3. Banerjee, A., 2021. *Getting Started with ESP32 Development*. Packt Publishing.
4. Datasheet - ACS712 Current Sensor, 2019. *Allegro Microsystems*. [online] Available at: https://www.sparkfun.com/datasheets/Components/General/ACS712.pdf [Accessed 25 Jun. 2025].
5. Dallas Semiconductor, 2021. *DS18B20 Programmable Resolution 1-Wire Digital Thermometer*. [pdf] Available at: https://datasheets.maximintegrated.com/en/ds/DS18B20.pdf [Accessed 25 Jun. 2025].
6. Espressif Systems, 2022. *ESP32-WROOM-32 Datasheet*. [pdf] Available at: https://www.espressif.com/sites/default/files/documentation/esp32-wroom-32_datasheet_en.pdf [Accessed 25 Jun. 2025].
7. GeeksforGeeks, 2023. *How to Send Sensor Data from ESP32 to MySQL Database using PHP*. [online] Available at: https://www.geeksforgeeks.org/how-to-send-sensor-data-from-esp32-to-mysql-database-using-php/ [Accessed 25 Jun. 2025].
8. Gonzalez, R., 2020. *Designing Embedded Systems with ESP32: A Practical Approach*. McGraw-Hill Education.
9. Ionic Framework, 2024. *Ionic Developer Documentation*. [online] Available at: https://ionicframework.com/docs [Accessed 25 Jun. 2025].
10. Karami, H. and Saidi, A., 2020. *A review on battery management system design and implementation*. Energy Reports, 6, pp.500–513.
11. Leaflet.js, 2024. *An Open-Source JavaScript Library for Mobile-Friendly Interactive Maps*. [online] Available at: https://leafletjs.com [Accessed 25 Jun. 2025].
12. Mehta, H., 2022. *Internet of Things (IoT) Projects with ESP32 and ESP8266*. Maker Media.
13. Microchip Technology, 2023. *NEO-6M GPS Module Datasheet*. [pdf] Available at: https://www.u-blox.com/sites/default/files/NEO-6_DataSheet_(GPS.G6-HW-09005).pdf [Accessed 25 Jun. 2025].

14. Mohanty, S., 2019. *Smart Battery Management Systems: A Case Study using IoT and Cloud*. Journal of Electrical Systems and Information Technology, 6(2), pp.265–276.
15. Open Source Hardware Community, 2024. *4x4 Matrix Keypad Interfacing with Arduino/ESP32*. [online] Available at: https://circuitdigest.com/microcontroller-projects/interfacing-4x4-keypad-with-arduino [Accessed 25 Jun. 2025].
16. PHP Manual, 2024. *MySQLi and PDO for Web-Based Database Access*. [online] Available at: https://www.php.net/manual/en/book.mysqli.php [Accessed 25 Jun. 2025].
17. Singh, A. and Prasad, R., 2021. *Design and Implementation of a Wireless Battery Monitoring System Using ESP32 and Mobile App*. International Journal of Computer Applications, 183(42), pp.19–26.
18. Tech Explorations, 2020. *Mastering ESP32 with Arduino IDE*. [eBook] Available at: https://techexplorations.com [Accessed 25 Jun. 2025].
19. W3Schools, 2024. *PHP and MySQL Tutorial*. [online] Available at: https://www.w3schools.com/php/php_mysql_intro.asp [Accessed 25 Jun. 2025].
20. Zhang, Y., Liu, J. and Wang, H., 2021. *Battery Parameter Estimation and Condition Monitoring for Smart IoT-Based BMS*. Journal of Energy Storage, 40, pp.102–112.