Faculty of Engineering
Department of Electrical and Electronics Engineering

EE 402 FINAL REPORT

Spring 2025

BATTERY HEALTH AND PERFORMANCE MANAGEMENT SYSTEM FOR LITHIUM-ION BATTERIES

Submitted by

Seyit Kubilay Uluçay
S029261

Atahan Dikeç
S024983

Supervisor

Dr. Hamza Makhamreh

APPROVAL PAGE

Faculty of Engineering
Department of Electrical and Electronics Engineering

EE 402 FINAL REPORT

Spring 2025

Seyit Kubilay Uluçay
Atahan Dikeç

BATTERY HEALTH AND PERFORMANCE MANAGEMENT SYSTEM FOR LITHIUM-ION BATTERIES

Jury Members:

Supervisor	:	**Dr. Hamza Makhamreh**	______________________
Jury Member 1	:	**Asst. Prof. Çağatay Edemen**	______________________
Jury Member 2	:	**Asst. Prof. Umut Başaran**	______________________

ABSTRACT

This report covers the study and early development of a Battery Management System for Lithium-ion batteries. These batteries are widely used in electric vehicles, renewable energy storage, and portable electronics because they have high energy density and are efficient. A BMS is very important to make sure these batteries are safe, reliable, and perform well. This is done by watching important data like voltage, current, and temperature. The BMS also plays a big part in balancing the cells to stop problems like too much charging, getting too hot, and losing capacity.

Key methods for guessing the battery's state, such as Coulomb Counting, Voltage Monitoring, were looked at. These methods were studied to understand how accurate they are, how complex they are, and if they work well when conditions change quickly. For cell balancing, both passive and active methods were compared.

The proposed system design uses an STM32F407 Discovery Board for building the first prototype. It includes important parts like TMP36GZ temperature sensors for checking temperature, and INA333 amplifiers with shunt resistors for measuring current and capacitor voltages. A custom PCB was designed to hold these parts. A special control algorithm was developed to manage the active cell balancing. Simulation results showed that this algorithm works well to balance cells during both charging and discharging.

While the main ideas of the design have been tested with simulations, the actual hardware has not yet been fully tested for long periods. This report explains the work done so far, the design of the BMS, and the good results from simulations. Future work will include more hardware testing and making the balancing faster.

ACKNOWLEDGMENT

We want to express our great appreciation to Dr. Hamza Makhamreh and Mehmet Karakoç for their extensive efforts in making this project a success.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

LIST OF SYMBOLS AND ABBREVIATIONS

ADC	Analog-to-Digital Converter
BMS	Battery Management System
C_1, C_2, C_3	Voltage of Balancing Capacitor 1, 2, 3
`chargeThreshold`	Algorithm threshold: minimum voltage difference to start balancing
`epsilon`	Algorithm tolerance: for checking if capacitor is fully charged
I	Current
IC	Integrated Circuit
KiCad	Software Tool
Li-ion	Lithium-ion
MATLAB	Software Tool
NTC	Negative Temperature Coefficient
OpAmp	Operational Amplifier
PCB	Printed Circuit Board
PWM	Pulse Width Modulation
R	Resistance
$S_1 \ldots S_8$	State of Switches 1 through 8 (0 = OFF, 1 = ON)
Simscape	MATLAB Toolbox
Simulink	Software Tool
SoC	State of Charge
SoH	State of Health
V	Voltage
V_1, V_2, V_3, V_4	Voltage of Cell 1, Cell 2, Cell 3, Cell 4
ΔV_{12}, ΔV_{23}, ΔV_{34}	Absolute voltage difference between adjacent cells

1 INTRODUCTION

Lithium-ion batteries are very common today. We use them in many things, like phones, electric cars, and for storing energy from any source. They are popular because they hold a lot of energy and last a long time. But, Li-ion batteries need careful management. Problems like charging too much, using too much power, getting too hot, or having cells with different voltage levels can be dangerous and make the battery age faster. To handle these problems, we need a good Battery Management System. A BMS watches important things like voltage, current, and temperature. It also helps estimate the battery's condition and protects the battery pack.

Our project is about building a BMS for Li-ion batteries. In the EE401 report, we researched BMS topics. We learned about how to estimate the battery state and different ways to balance the cells. Now, in this report, we are explaining the steps in the building phase of the BMS system.

The main goal of the project is to make a working BMS prototype. Our main tasks are:

- Build the Hardware: Design and make a custom circuit board (PCB). Put the STM32F407 Discovery Board, sensors, and balancing parts on it.
- Make the Balancing System: Build the active balancing circuit using capacitors and special switches.
- Write the Code: Program the STM32F407 board. This includes writing the code for the Control Algorithm that controls the balancing switches using sensor readings.
- Test Everything: Put all the parts together and test if the BMS works correctly. We need to check if it measures things accurately and balances the cells.
- Plan is to use the STM32F407 discovery board first. Capacitor based balancing circuit will be built. The control code will be written based on our MATLAB simulations.

2 METHODOLOGY

This section explains the steps taken and the tools used to build the BMS. Overall work involves designing hardware, creating a control mechanism, and testing the system.

2.1 Problem Formulation

The main problem is managing Li-ion batteries correctly. Li-ion batteries are used everywhere, but they can be risky if not handled properly. If they are charged too much, used too much, got too hot, or if the different cells inside the battery pack have uneven voltages, the battery can get damaged. This damage reduces the battery's life and can even lead to dangerous situations like fires. Standard battery systems might not handle these issues well. So, the engineering problem is to create a reliable system that prevents these dangers and helps the battery work better and last longer.

2.2 Proposed Solution

To solve the problems mentioned before, the proposed solution is to build a custom BMS. This system is designed to manage a battery pack of four Li-ion cells connected in series. The main goal of our BMS is to monitor the battery pack continuously to prevent the occurrence of overcurrent, overcharge, or overheating problems while performing active cell balancing to keep all cells at similar voltage levels. All of the controls and management will be held in the STM32 board, so the proposed system will be able to be configured as required or desired.

To make the voltage differences between each cell smaller, energy will be moved between cells using capacitors. Capacitors are small, so they will need to make the voltage differences smaller over many cycles. So, the capacitors will charge and then discharge many times to move the charge. For capacitor-based active balancing, a common way to control charging and discharging is by using switches. These switches connect the capacitor to either the higher voltage cell for charging or the lower voltage cell for discharging. This control is mostly done using two PWM signals. One PWM signal has its timing shifted, starting halfway through the period of the other signal to manage the charge and discharge steps. This approach is used by MathWorks as a Simulink example [5]

However, this report proposes a different approach to the switch control problem by creating an algorithm to control switches of the capacitor, which takes cell and capacitor voltages as inputs and controls the switches while maintaining the state of batteries in a stable state.

2.3 Components

The proposed BMS has several key parts working together.

2.3.1 Safety Switch

BD135: The BD135 NPN transistor, shown in the figure 1 was chosen for its reliability and sufficient current-handling capability in low-side switching applications.

Figure 1: BD135 photograph in real life [1].

It is used in the circuit for grounding control in a battery protection mechanism as shown in the figure 2. This transistor is controlling the path between main negative and ground, particularly useful in hardware-based disconnection of the system's negative line under fault conditions.

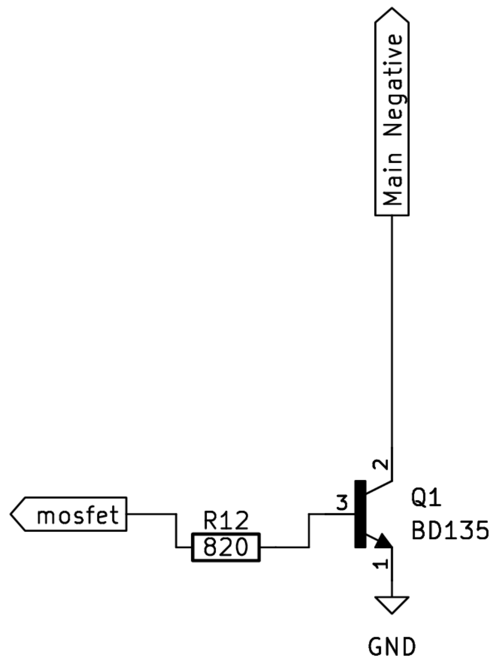

Figure 2: BD135 transistor for safety.

2.3.2 Operational Amplifiers

The INA333 instrumentation amplifier, shown in the figure 3, was utilized for its high common-mode rejection ratio and low offset voltage, making it ideal for accurate differential voltage measurements across a shunt resistor to calculate battery current and capacitor voltages.

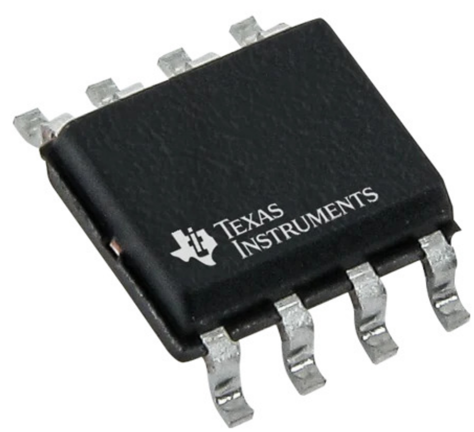

Figure 3: INA333 photograph in real life [2].

In the figure 4 configuration of the current reading system configuration based on the INA333 OpAmp.

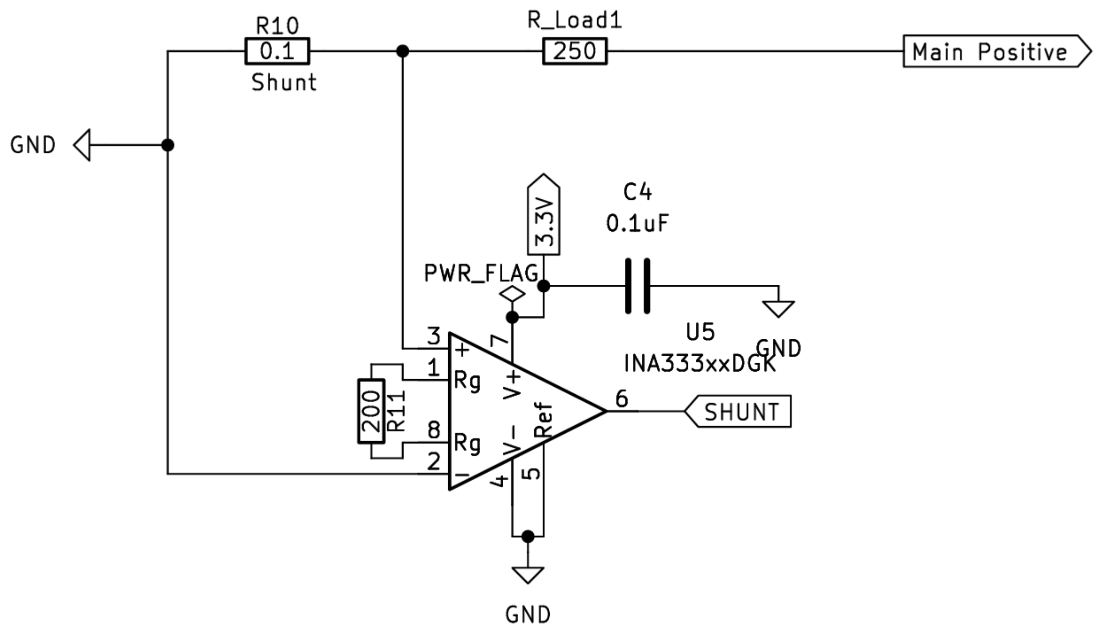

Figure 4: Amplification at the shunt resistor.

2.3.3 Capacitor Switches

AQV252G: The AQV252G Photomos solid-state relay transistor, shown in the figure 5, was chosen due to its low on-resistance, high isolation voltage, and compact form, making it suitable for safe and efficient switching in battery protection systems.

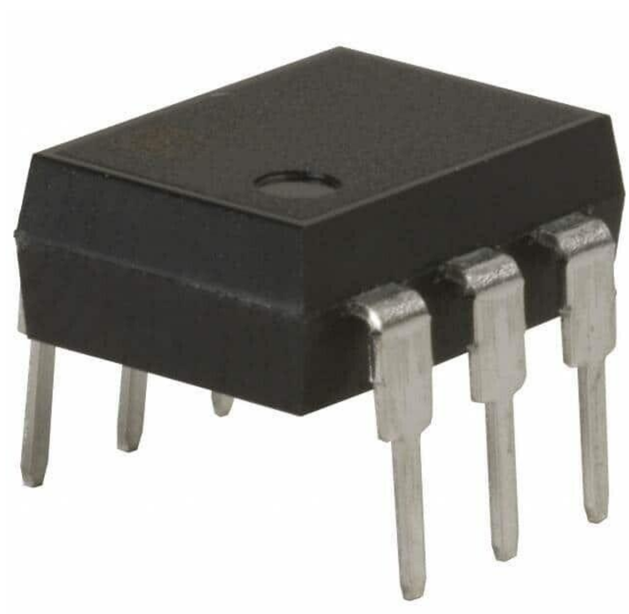

Figure 5: AQV252G Real life image [3].

System configuration of the AQV252G Photomos in the PCB is shown in the figure 6.

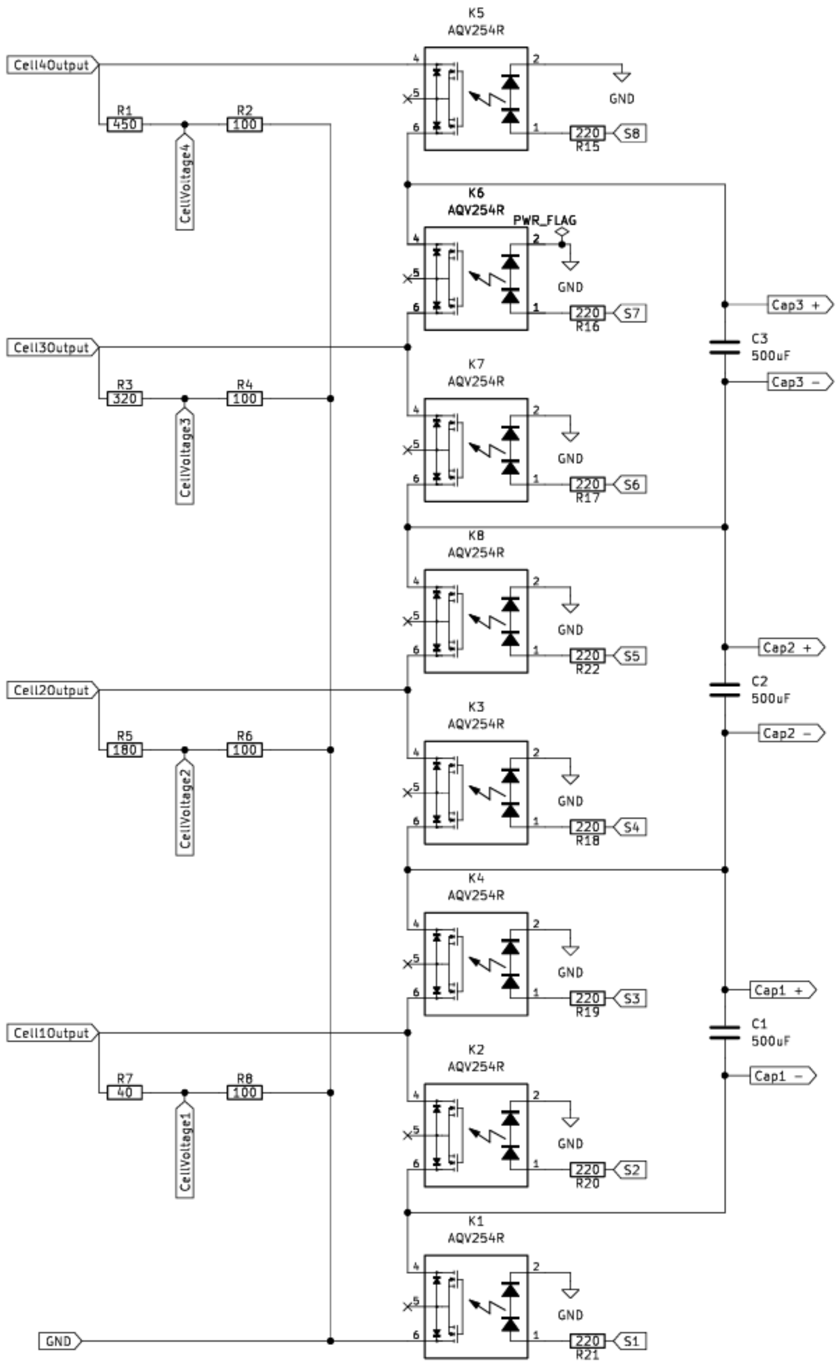

Figure 6: AQV252G connections.

The AQV252G photomos is supplied energy from the pin 1 and connected to GND from the pin 2, this connection runs the photomos, with providing energy to photomos it is possible to turn on or off the photomos, which controls the connection between pins 6 and 4 as shown in the figure 6.

2.3.4 Temperature Readings

The temperature of the battery cells is an important factor for safety and performance. In this system, Negative Temperature Coefficient NTC thermistors (specifically TMP36GZ model as shown in the figure 7) are planned to be used for reading cell temperatures.

Figure 7: TMP36GZ Real image [4].

These NTC thermistors will be directly connected to analog input pins on the STM32 board. The STM32 will then convert the analog voltage readings from the thermistors into temperature values. This information will be used by the BMS to monitor for overheating or overcooling conditions, helping to keep the battery operating within safe temperature limits. The connections for two thermistors, Thermistor1 and Thermistor2, are shown in the schematic in Figure 8.

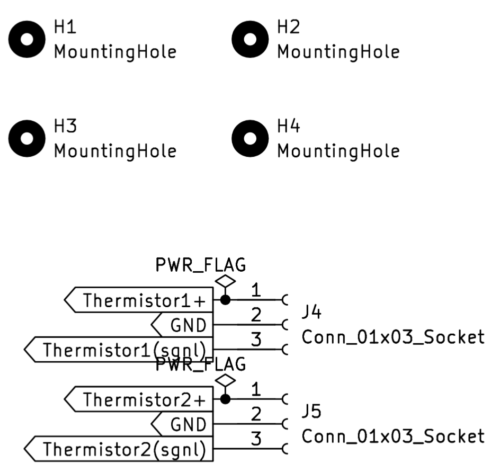

Figure 8: Mounting Holes and Thermistor connections at the schematic.

2.3.5 Central Controller

The main control unit is an STM32F407 Discovery Board, which is shown in the figure 9. This board reads all the sensor data, runs the control algorithm, and decides when and how to balance the cells.

The STM32F407G Discovery board was selected for its high-performance, wide range of peripherals, and lots of ADC channels, which are critical for handling simultaneous voltage, temperature, and current measurements in real-time. Specific pin connections on the STM32F407G boards were made based on signal requirements and ADC channel availability. CellVoltage1 through CellVoltage4 were connected to analog-capable pins on U1 to allow direct measurement of individual cell voltages via the internal ADCs. The BD135 transistor control line is assigned to a GPIO pin to allow digital control over external switching elements such as transistors. On U2, the Thermistor1 signal and Thermistor2 signal inputs are connected to analog-capable pins to monitor battery temperature for safety and thermal management. Voltage Reader 1, 2, and 3 are also connected to analog inputs to enable voltage sensing at various points in the battery pack. The SHUNT signal is routed to an analog input as well, as it represents the differential voltage drop across a shunt resistor used for current measurement. Switches S1 to S8 are assigned to general GPIO pins with interrupt capability to provide manual or logic-based interaction with the system. This pin configuration ensures optimal use of the microcontroller's resources, balancing analog input needs with digital control and user interaction requirements.

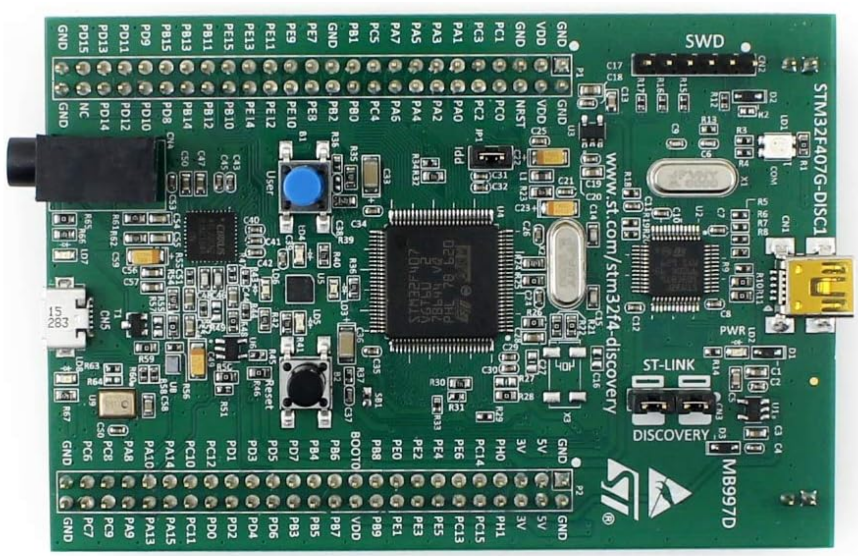

Figure 9: STM32F407 Discovery Kit

2.3.6 Simulations

Before designing the hardware, we performed simulations to test ideas we had planned earlier, especially the active balancing control algorithm. The idea for the simulation approach was initially based on an example found on the MathWorks website, called "Balance Battery Cells with Switched Capacitor Method" as shown in the figure 10. This example, built using Simscape Battery, showed how capacitor-based active balancing could be modeled and simulated in Simulink [5].

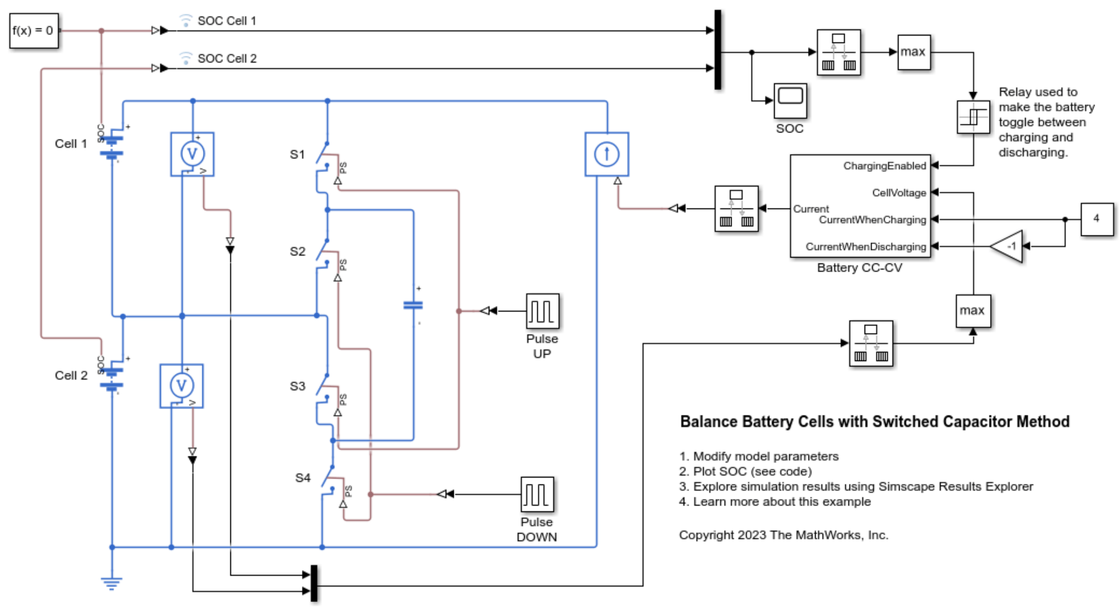

Figure 10: Mathworks 2-Cell Capacitor Based Active Balancing

Mathworks example was a great starting point because it provided a 2-cell example for capacitor-based active balancing. However, some changes were needed for our specific project.

- **Number of Cells:** The example used only two cells. Our project requires managing four cells, so the simulation model was expanded to include four battery cell blocks connected in series.
- **Control Method:** The original MathWorks example controlled the balancing switches using simple PWM signals. For our BMS, a more intelligent control method is needed. Therefore, the PWM control was replaced with our custom ControlAlgorithm, which makes decisions based on real-time voltage measurements from the cells and capacitors.

We designed MATLAB/Simulink to build a model of our 4-cell battery pack and the capacitor-based balancing circuit with Simscape toolbox as shown in the figure 11.

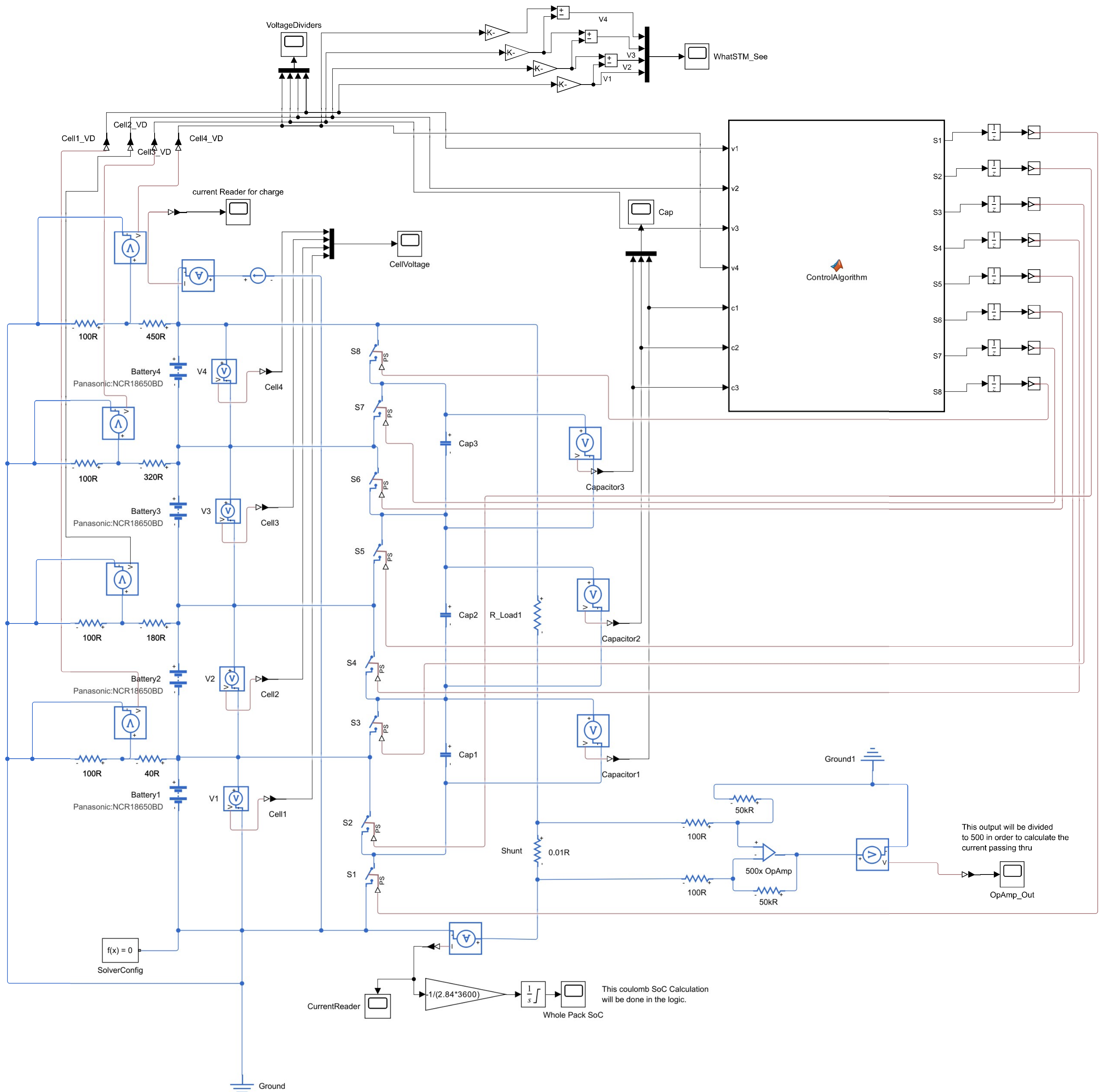

Figure 11: Simulation Overview

In our simulation model:

- Battery components are representing the Panasonic NCR18650BD cells, which are planned for use in the hardware, were choosen.
- The 4 cells were set up with different starting SoC levels (like 100%, 90%, 80%, and 70%) so that an imbalance was created for the BMS to fix.
- The balancing circuit, including three capacitors set to 500 microfarads based on tests and the eight switches, was included in the model.
- The ControlAlgorithm logic was implemented to control the switches based on the simulated cell and capacitor voltages.

2.3.7 Reading Values In Simulation

In the simulation current passing through the load, individual cell voltages, capacitor voltages are getting red by the STM32 board.

Reading Current The current passing through the load is also passing through shunt resistor as shown in the figure 12 and the shunt resistor has small amount of resistance in it, for example 0.1Ω. But still this resistance causes small amount of voltage drop across the shunt resistor. This voltage drop is after amplified by INA333 OpAmp with 500 times and this signal is sent to STM32 board. Since the resistance value is so small inside the shunt resistor, the voltage drop across the resistor is can be closely estimated as current passing through the resistor because of the Ohms' law $V = I * R$, when R value is too low, it can be negligible and we can assume $V = I$ in this case.

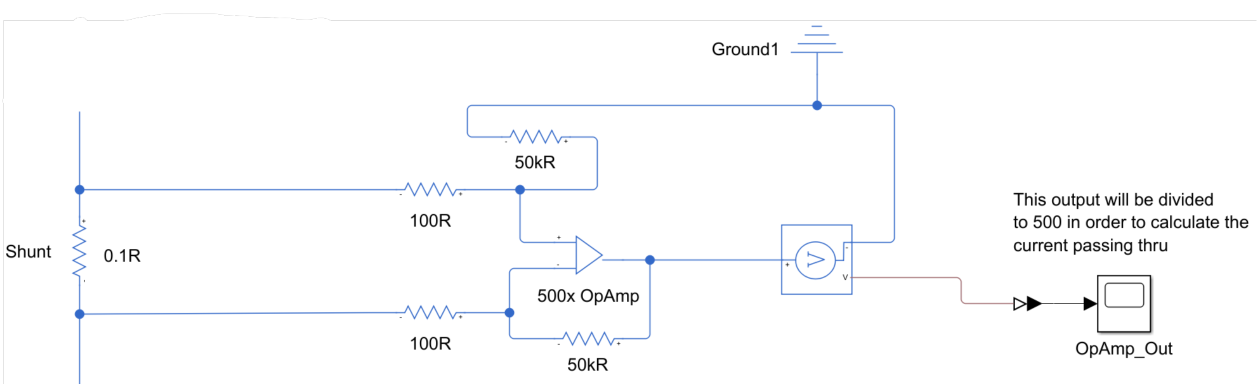

Figure 12: Current reader shunt resistor connection.

Reading Voltages The individual cell voltages are can be directly read by ADC of the STM32 board, however 4.2 volts are the max limit of the Li-ion cells so they can be exceed the 3.3V limit of the ADC of STM32 board and damage the board. So one of the most basic solution to read voltages are using voltage dividers as shown in the figure 13 and then map these values inside the STM32 board to correct values.

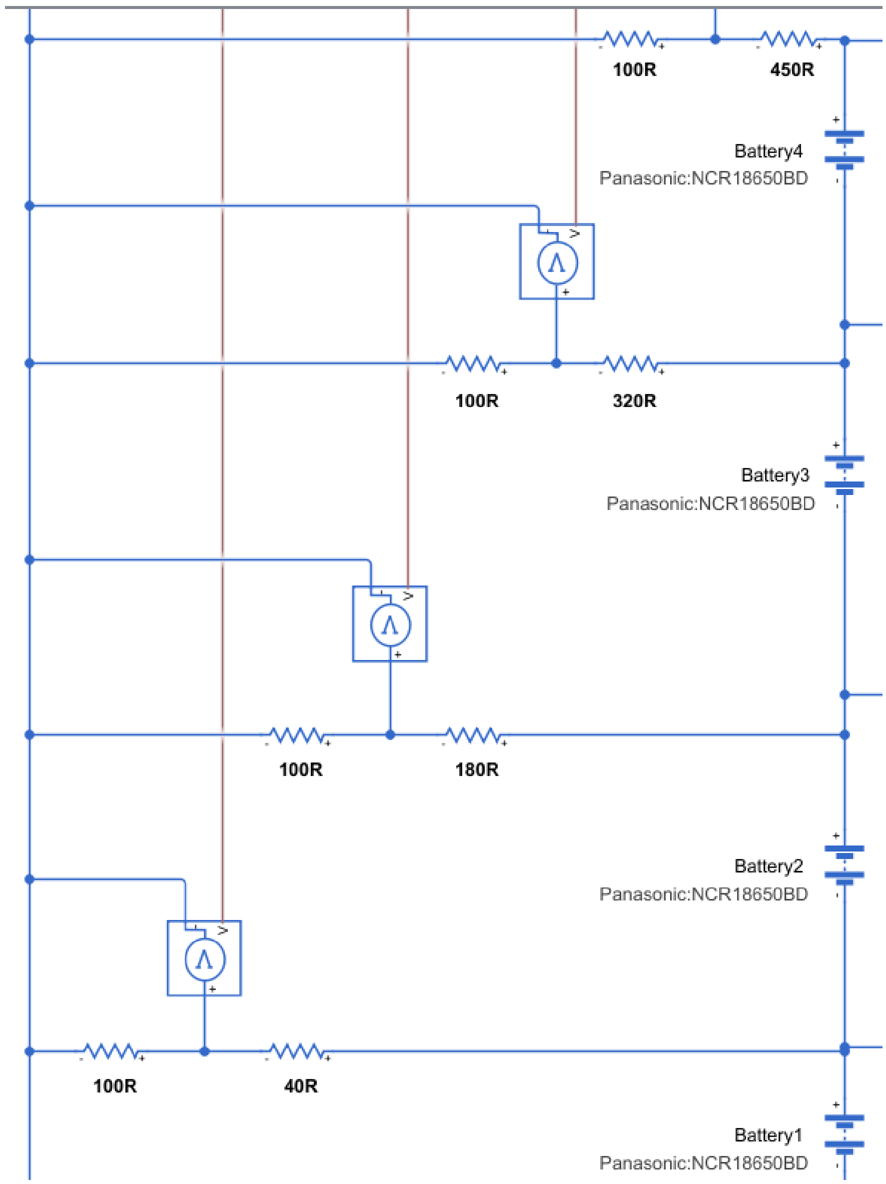

Figure 13: Voltage divider circuit.

The capacitor voltages are also now read by INA333 OpAmps for control purposes. Moreover, the temperature value of the cells will be read with TMP36GZ NTC thermistors and they will be directly connected to STM32 board.

2.3.8 Control Algorithm

The ControlAlgorithm shown in below, developed in MATLAB/Simulink, Control Algorithm is created inside the function block which is actually representing the behaviour of the STM32 board with data values and controls the balancing switches. At the Listing 1 The function takes the voltages of the four cells V1 to V4 and the three balancing capacitors C1, C2 and C3 as inputs. It outputs the control signals 0 for OFF, 1 for ON for the eight switches S1 to S8.

This MATLAB function implements a control algorithm for active cell balancing in a BMS, targeting voltage equalization across multiple cells. The function takes four cell voltage inputs (v1 through v4) and three capacitor voltages (c1 through c3) used for charge transfer between cells. It outputs eight control signals (S1 to S8) which likely correspond to switch states in the balancing circuit.

A set of persistent variables (balancingCap, pulsesRemaining, toggleState, Nval) is used to retain the system's state across function calls which set on the Listing 1. These variables help ensure continuity in the balancing process. Initially, the algorithm sets up the state to indicate that no capacitor is actively balancing (balancingCap = 0) and no toggling is ongoing. toggleState manages whether the balancing capacitor is in a charging or discharging phase. The N_val parameter defines how many full toggling cycles (charge/discharge) are performed before the system re-evaluates the voltage differences and balancing status.

The constants THRESHOLD_TO_PICK_NEW_PAIR and TARGET_DV_FOR_ACTIVE_PAIR define the criteria for initiating new balancing actions and terminating current ones. Specifically, a new cell pair will be selected for balancing if their voltage difference exceeds a threshold (10 mV), and balancing on the current pair will persist until the voltage difference is reduced to a desired target (10 mV). The algorithm is designed to perform persistent balancing, meaning it does not switch cell pairs too frequently, which could lead to instability or inefficiency.

Overall, the function provides a systematic approach to dynamic and state-aware active balancing, preserving both charge uniformity and operational efficiency in the battery pack.

```matlab

function [S1,S2,S3,S4,S5,S6,S7,S8] = ControlAlgorithm(v1,v2,v3,v4, c1, c2, c3)
% ControlAlgorithm.m (Modified for persistent balancing until target)

persistent balancingCap pulsesRemaining toggleState N_val
if isempty(balancingCap)
    balancingCap    = 0;  % 0=no active cap, 1=C1, 2=C2, 3=C3
    pulsesRemaining = 0;  % half strokes left
    toggleState     = 0;  % 0=charge, 1=discharge
    N_val           = 10; % Number of full pulses per "commitment" before checking pulsesRemaining
end

% --- Algorithm Constants ---
THRESHOLD_TO_PICK_NEW_PAIR = 0.010; % (10mV) Min difference to initiate balancing on a NEW pair.
TARGET_DV_FOR_ACTIVE_PAIR  = 0.010; % (10mV) Balance the active pair until its |dV| is <= this value.
```

Listing 1: MATLAB Control Algorithm Part 1.

This part which is shown in Listing 2 reconstructs individual cell voltages from cumulative voltage measurements. Each Vcell is calculated by subtracting the scaled contribution of previous cells, using calibration constants likely based on voltage dividers or ADC scaling. This enables accurate monitoring of each cell individually in the series-connected battery pack.

```matlab
%--- 1. Reconstruct the actual cell voltages ---
Vcell1 = 1.40 * v1;
Vcell2 = 2.80 * v2 - 1.40 * v1;
Vcell3 = 4.20 * v3 - 2.80 * v2;
Vcell4 = 5.50 * v4 - 4.20 * v3;
```

Listing 2: MATLAB Control Algorithm Part 2.

This section at the Listing 3 calculates the voltage differences between adjacent cells to identify imbalance. It then takes the absolute values of these differences to evaluate the magnitude of imbalance regardless of polarity. These values are used to determine whether balancing is needed between any pair of cells.

```
%--- 2. Compute adjacentcell differences ---
dV12 = Vcell2 - Vcell1;
dV23 = Vcell3 - Vcell2;
dV34 = Vcell4 - Vcell3;
abs_dV12 = abs(dV12);
abs_dV23 = abs(dV23);
abs_dV34 = abs(dV34);
```

Listing 3: MATLAB Control Algorithm Part 3.

This section in Listing 4 defines the decision-making logic for initiating and continuing the balancing process. It starts by checking whether a capacitor is currently active. If so, it evaluates whether the voltage difference in the corresponding cell pair has fallen below the target threshold. If the target is met, balancing for that pair stops. Otherwise, if the pulse count (pulsesRemaining) has expired, it is reset to continue balancing.

If no capacitor is currently active, the algorithm compares all adjacent cell voltage differences and selects the pair with the maximum imbalance, provided it exceeds a predefined threshold. That pair is then assigned for balancing with a fresh pulse quota, and the system is reset to begin a new balancing cycle. This approach ensures efficient and persistent balancing only when necessary.

```

%--- 3. Decision Logic ---
S_internal = zeros(1,8);
% Part A: Check if an already active capacitor has met its target
if balancingCap ~= 0
    current_active_dV_abs = 0;
    if balancingCap == 1
        current_active_dV_abs = abs_dV12;
    elseif balancingCap == 2
        current_active_dV_abs = abs_dV23;
    elseif balancingCap == 3
        current_active_dV_abs = abs_dV34;
    end

    if current_active_dV_abs <= TARGET_DV_FOR_ACTIVE_PAIR
        % This active pair has reached its target. Stop balancing it.
        balancingCap = 0;    % This will trigger picking a new pair in Part B.
        pulsesRemaining = 0; % Ensure no more pulses for this now-stopped cap.
```

```
    else
        % Active pair still needs balancing (its |dV| > TARGET_DV_FOR_ACTIVE_PAIR).
        % If its pulse quota for the current N_val commitment ran out, renew it.
        if pulsesRemaining == 0
            pulsesRemaining = 2 * N_val;
        end
        % It will proceed to execute a pulse in Step 4.
    end
end

% Part B: If no capacitor is active (either initially, or previous one met its target)
if balancingCap == 0
    all_abs_dVs = [abs_dV12, abs_dV23, abs_dV34];
    [maxDV_overall, idx] = max(all_abs_dVs);

    if maxDV_overall > THRESHOLD_TO_PICK_NEW_PAIR
        % A pair is found that needs balancing.
        balancingCap    = idx;       % 1, 2, or 3
        pulsesRemaining = 2 * N_val; % Give it a fresh set of pulses
        toggleState     = 0;         % Always start a new balancing sequence with a "charge" stroke
    else
        % No pair meets the criteria to START balancing. All are balanced enough.
        balancingCap = 0;    % Ensure it's off
        pulsesRemaining = 0;
    end
end
```

Listing 4: MATLAB Control Algorithm Part 4.

This part of the code that is shown in the Listing 5 handles the execution of one half-cycle (charge or discharge) of the active balancing operation using capacitor-based charge shuttling. Depending on the selected cell pair (balancingCap), the algorithm compares the voltage difference to determine the direction of charge transfer. It then activates the appropriate switches (S_internal entries) to either charge the balancing capacitor from the higher-voltage cell or discharge it to the lower-voltage cell.

The switching alternates between charge and discharge phases by toggling toggleState on each call, while pulsesRemaining tracks how many half-cycles are left. If no balancing is currently active, all switch signals remain off. This structure ensures controlled, directional energy transfer between imbalanced cells.

```

%--- 4. If a cap is active, execute one  h a l f stroke  ---
if balancingCap ~= 0 % A cap is selected and should be active
    if balancingCap == 1
        if dV12 > 0 % Cell2 > Cell1     shuttle from  C 2 C1
            if toggleState == 0; S_internal(2) = 1; S_internal(4) = 1; % Charge C1 from Cell2 (via S2, S4)
            else;                S_internal(1) = 1; S_internal(3) = 1; % Discharge C1 to Cell1 (via S1, S3)
            end
        else % Cell1 > Cell2     shuttle from  C 1 C2
            if toggleState == 0; S_internal(1) = 1; S_internal(3) = 1; % Charge C1 from Cell1
            else;                S_internal(2) = 1; S_internal(4) = 1; % Discharge C1 to Cell2
            end
        end
    elseif balancingCap == 2
        if dV23 > 0 % Cell3 > Cell2     shuttle  C 3 C2
            if toggleState == 0; S_internal(4) = 1; S_internal(6) = 1;
            else;                S_internal(3) = 1; S_internal(5) = 1;
            end
        else % Cell2 > Cell3     shuttle  C 2 C3
            if toggleState == 0; S_internal(3) = 1; S_internal(5) = 1;
            else;                S_internal(4) = 1; S_internal(6) = 1;
            end
        end
    elseif balancingCap == 3
        if dV34 > 0 % Cell4 > Cell3     shuttle  C 4 C3
            if toggleState == 0; S_internal(6) = 1; S_internal(8) = 1;
            else;                S_internal(5) = 1; S_internal(7) = 1;
            end
        else % Cell3 > Cell4     shuttle  C 3 C4
            if toggleState == 0; S_internal(5) = 1; S_internal(7) = 1;
            else;                S_internal(6) = 1; S_internal(8) = 1;
            end
        end
```

```
    end

    pulsesRemaining = pulsesRemaining - 1;
    toggleState     = 1 - toggleState;
else
    % No balancing action this cycle, S_internal remains all zeros.
end
```

Listing 5: MATLAB Control Algorithm Part 5.

This part of the code in the Listing 6 segment is specifically designed to prevent short-circuit conditions. It operates on a state vector S (representing switch/contactor states) by sequentially applying six "no-short" rules. Each rule, implemented using conditional if statements (for example, S(8) && S(6)), checks for specific concurrent switch activations that could lead to a short. If such a hazardous combination is detected, the algorithm proactively deactivates (sets to 0) other designated switches within the S vector. The explicitly enforced order of these rule evaluations is critical for ensuring deterministic fault prevention and maintaining the electrical safety of the battery system.

```
%--- 5. Enforce the six  n o short  rules, in exactly this order (on S_internal)
S = S_internal;
if (S(8) && S(6)); S(7) = 0; S(5) = 0; end
if (S(7) && S(5)); S(8) = 0; S(6) = 0; S(4) = 0; S(2) = 0; end
if (S(6) && S(4)); S(8) = 0; S(7) = 0; S(5) = 0; S(3) = 0; S(2) = 0; S(1) = 0; end
if (S(5) && S(3)); S(8) = 0; S(7) = 0; S(6) = 0; S(4) = 0; S(2) = 0; S(1) = 0; end
if (S(4) && S(2)); S(7) = 0; S(5) = 0; S(3) = 0; S(1) = 0; end
if (S(3) && S(1)); S(4) = 0; S(2) = 0; end
```

Listing 6: MATLAB Control Algorithm Part 6.

This part of the code segment at the Listing 7 is the final step. It takes a list of switch commands, which is stored inside a variable called S. The code then separates each command from this list and gives it its own name, from S1 up to S8. For example, the first command in the S list becomes S1, the second command becomes S2, and so on. These individual S1 to S8 variables are the final 'on' or 'off' signals for each of the eight switches. This makes it easy to use these signals to control the actual switches in the hardware, or to send this information to other parts of the system, or just to see what the final decision for each switch is.

```
%--- 6. Unpack S back into the eight outputs ---
S1 = S(1); S2 = S(2); S3 = S(3); S4 = S(4);
S5 = S(5); S6 = S(6); S7 = S(7); S8 = S(8);

end
```

Listing 7: MATLAB Control Algorithm Part 7.

2.3.9 Code Algorithm

The MATLAB-based Control Algorithm, detailed in Listings 1 through 7, manages the active cell balancing process. Its operation is based on several key steps that are executed repeatedly:

1. **State Retention and Initialization (Listing 1):** Variables are used to remember the balancing state between function calls. These include:

   - `balancingCap`: This shows which capacitor (C1, C2, or C3) is currently active. A value of 0 means no capacitor is active.
   - `pulsesRemaining`: This counts how many more charge or discharge actions (half-strokes) are left for the currently active capacitor before its status is checked again.
   - `toggleState`: This decides if the active capacitor should next charge (0) or discharge (1).
   - `N_val`: This sets how many full charge/discharge cycles are done by an active capacitor before the system checks if it should continue with that same capacitor or look for a new cell pair to balance.

   Two main voltage difference levels are also set: `THRESHOLD_TO_PICK_NEW_PAIR` (10 mV) is the minimum difference needed to start balancing a new pair of cells. `TARGET_DV_FOR_ACTIVE_PAIR` (10 mV) is the level to which the voltage difference of an already active pair should be reduced.

2. **Cell Voltage Calculation (Listing 2):** The actual voltage of each individual cell (Vcell1, Vcell2, Vcell3, Vcell4) is calculated. This is done using the input voltage readings (v1, v2, v3, v4), which are likely from voltage dividers measuring sums of cell voltages. Specific constants (1.40, 2.80, etc.) are used to find each cell's true voltage.

3. **Voltage Difference Calculation (Listing 3):** The voltage differences between cells that are next to each other (dV12, dV23, dV34) are found. The absolute (positive) values of these differences are also calculated to see how big the imbalance is.

4. **Balancing Decision Making (Listing 4):** This part decides if balancing is needed and which cells to balance.

   - If a capacitor is already balancing a pair of cells: The voltage difference of this active pair is checked. If this difference is now less than or equal to `TARGET_DV_FOR_ACTIVE_PAIR`, balancing for this pair is stopped. The system will then look for a new pair. If the target is not met but the `pulsesRemaining` for this capacitor has reached zero, it means the capacitor has completed its set number of actions. So, `pulsesRemaining` is reset to allow it to continue balancing the same pair.
   - If no capacitor is currently balancing: The voltage differences of all cell pairs are compared. The pair with the largest difference is chosen. If this largest difference is greater than `THRESHOLD_TO_PICK_NEW_PAIR`, that pair is selected for balancing. The `balancingCap` variable is set to the chosen capacitor, `pulsesRemaining` is set for a new set of actions, and `toggleState` is set to start with charging the capacitor. If no pair has a big enough difference, no balancing is started.

5. **Executing a Balancing Action (Listing 5):** If a capacitor is chosen for balancing and `pulsesRemaining` is greater than zero, one balancing action (a half-stroke) is performed.

   - The direction of energy transfer is decided by looking at the voltage difference of the chosen cell pair (e.g., dV12).
   - If `toggleState` is 0 (charge phase): The correct switches (S1 to S8, stored in `S_internal`) are turned on to charge the capacitor from the cell with the higher voltage.
   - If `toggleState` is 1 (discharge phase): The switches are set to discharge the capacitor into the cell with the lower voltage.
   - After the action, `pulsesRemaining` is reduced by one, and `toggleState` is flipped for the next action.

   If no balancing is active, all switch commands in `S_internal` are kept off.

6. **Preventing Short Circuits (Listing 6):** A set of six rules is applied in a specific order. These rules check the planned switch commands (`S_internal`) to make sure no combination of ON switches would cause a short circuit in the battery system. If a risky combination is found, some switches are forced OFF to prevent the short.

7. **Setting Final Switch Outputs (Listing 7):** The final, safe switch commands from the internal list S are given to the output variables S1 through S8. These outputs are then used to control the actual hardware switches.

Until the ideal control algorithm is found, the algorithm has changed over 10 times from scratch. However, Matlab simulation with control algorithm run with different states and showed good performance, the most important two states are charging as shown in the figure 15 and discharging 14.

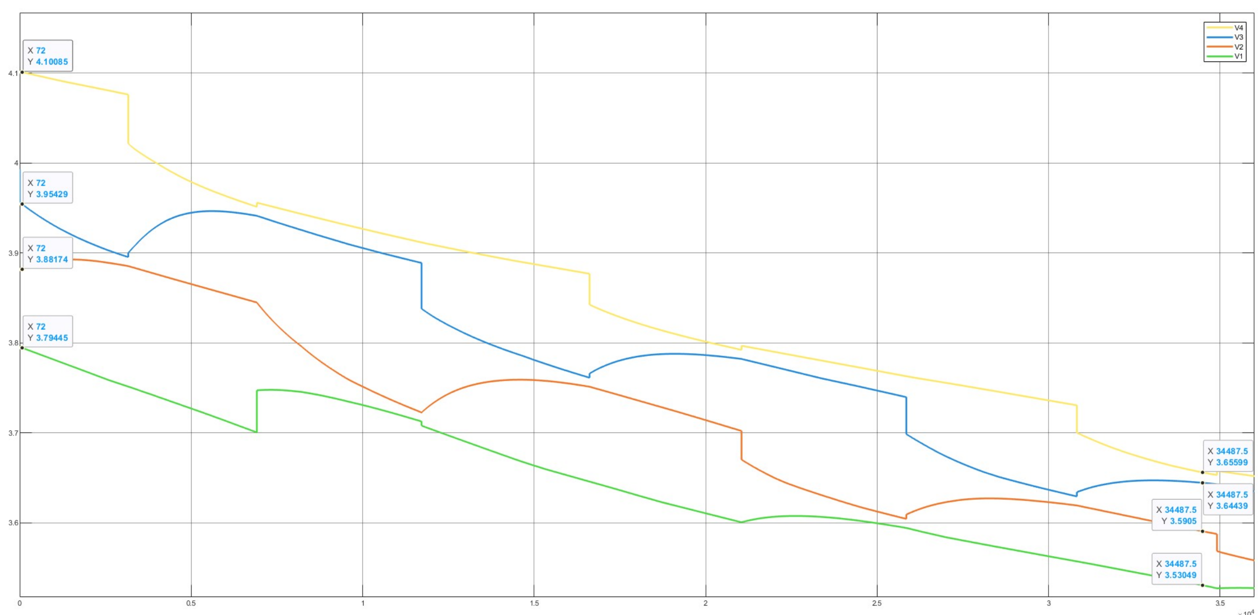

Figure 14: Cell Values of the battery while discharging the system with control algorithm.

The simulations showed that the Control Algorithm works as expected. When the simulated started with unbalanced cells as shown in the table 1, the algorithm detected the voltage differences. It then correctly turned the switches ON and OFF to charge and discharge the capacitors, moving energy between the cells. As the simulation ran, the voltage differences between the cells and the total imbalance between cells got smaller, as you can see in the table 2.

Table 1: Cell Voltages and Imbalance at an Early Discharging Stage

Time (s)	Cell 4 (V)	Cell 3 (V)	Cell 2 (V)	Cell 1 (V)	Sum of $\|\Delta V\|$ (mV)
0	4.101479	3.993070	3.880436	3.795459	306.02
5	4.101420	3.956361	3.880530	3.795401	306.02
10	4.101377	3.956205	3.880623	3.795330	306.05
15	4.101335	3.956049	3.880716	3.795259	306.08
20	4.101292	3.955894	3.880809	3.795188	306.10

Table 2: Cell Voltages and Imbalance at a Later Discharging Stage

Time (s)	Cell 4 (V)	Cell 3 (V)	Cell 2 (V)	Cell 1 (V)	Sum of $\|\Delta V\|$ (mV)
33390	3.665379	3.646941	3.598976	3.538581	126.80
33395	3.665329	3.646936	3.598937	3.538545	126.78
33400	3.665279	3.646930	3.598897	3.538510	126.77
33405	3.665229	3.646924	3.598858	3.538474	126.76
33410	3.665179	3.646919	3.598819	3.538438	126.74

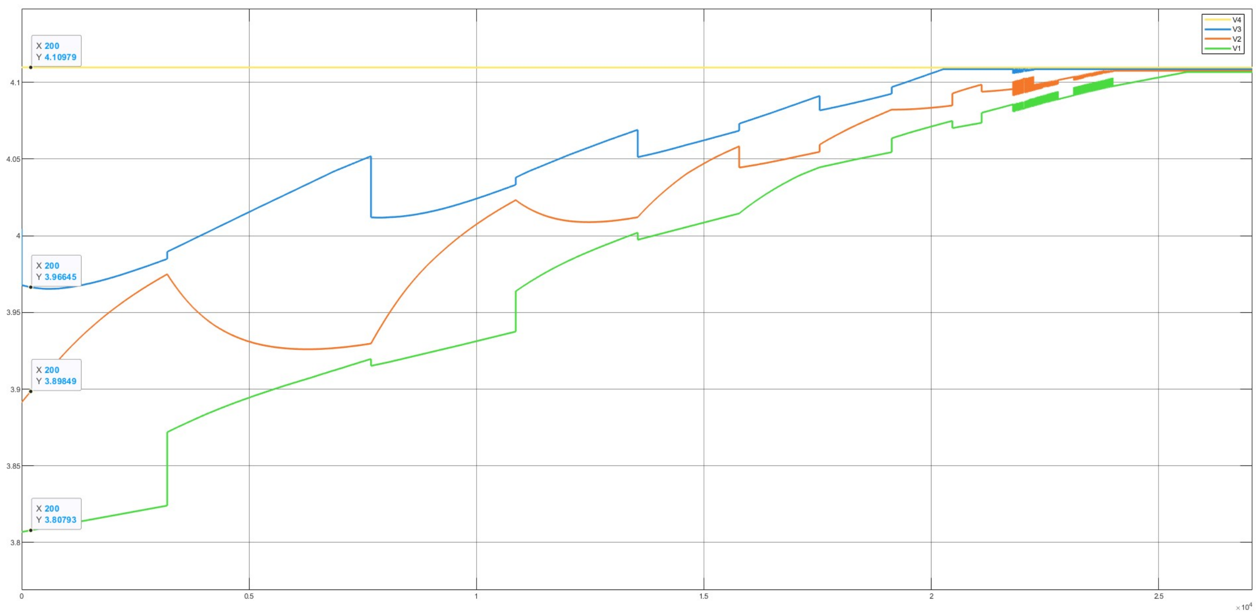

Figure 15: Cell Values of the battery while charging the system with control algorithm.

Then the same data and simulation run again but the major difference was now the cells were getting charged up with constant current while the balancing algorithm was running in the background.

Table 3: Cell Voltages and Imbalance at an Early Charging Stage

Time (s)	Cell 4 (V)	Cell 3 (V)	Cell 2 (V)	Cell 1 (V)	Sum of $\|\Delta V\|$ (mV)
0	4.109786	4.004540	3.891349	3.806844	302.94
5	4.109793	3.967767	3.891535	3.806884	302.91
10	4.109793	3.967728	3.891721	3.806911	302.88
15	4.109793	3.967689	3.891907	3.806938	302.86
20	4.109793	3.967651	3.892092	3.806965	302.83

Table 4: Cell Voltages and Imbalance at a Later Charging Stage

Time (s)	Cell 4 (V)	Cell 3 (V)	Cell 2 (V)	Cell 1 (V)	Sum of $\lvert\Delta V\rvert$ (mV)
27040	4.109666	4.108565	4.107561	4.106739	2.93
27045	4.109666	4.108565	4.107561	4.106739	2.93
27050	4.109666	4.108565	4.107561	4.106739	2.93
27055	4.109666	4.108565	4.107561	4.106739	2.93
27060	4.109666	4.108565	4.107561	4.106739	2.93

Table 3 displays cell voltages at an early phase of the charging and balancing process. The sum of absolute voltage differences, $\sum \lvert\Delta V\rvert$, starts at 302.94 mV and shows a slight decrease to 302.83 mV. This indicates that the balancing algorithm is beginning to address the initial, relatively large imbalance.

Significantly, Table 4, which records data much later in the charging process (around 27040 seconds), reveals that the $\sum \lvert\Delta V\rvert$ has been dramatically reduced to a constant 2.93 mV. The individual cell voltages are very close to each other.

Results confirmed that the capacitor-based balancing approach and our control logic could effectively balance the cells and reduces the voltage difference in both charging and discharging states. This gave us confidence to proceed with the hardware design based on this simulated system.

2.4 Custom PCB

We designed a custom PCB using KiCad to connect all the parts of our BMS together in one place, as shown in the figure 16. This board acts as a motherboard.
The main component mounted on this PCB is the STM32F407 Discovery Board (labeled U1A and U1B in the schematic). The PCB routes connections between the STM32 board and all other parts of the system:

- **Battery Connections:** The PCB has points to connect the four Li-ion battery cells (BT1+ to BT4+ shown on schematic).
- **Sensor Integration:** All the sensor circuits are built onto the PCB. This includes the voltage reading circuits which using INA333 amplifiers U6-U8 for each capacitor is connected ports PC0, PC2 and PA2, the current sensing circuit shunt resistor and INA333 amplifier U5 is

connected port PA0, and the connections for the NTC thermistors are connected to PE7 and PE9.

- **Balancing Circuit Integration:** The active balancing circuit, including the eight AQV252G photomos switches K1-K8 are connected to ports PA6, PA4, PD0, PD2, PE3, PE5, PD8 and PD15.
- **Power Distribution:** The PCB also distributes the necessary power 3.3V and Ground to all required components.

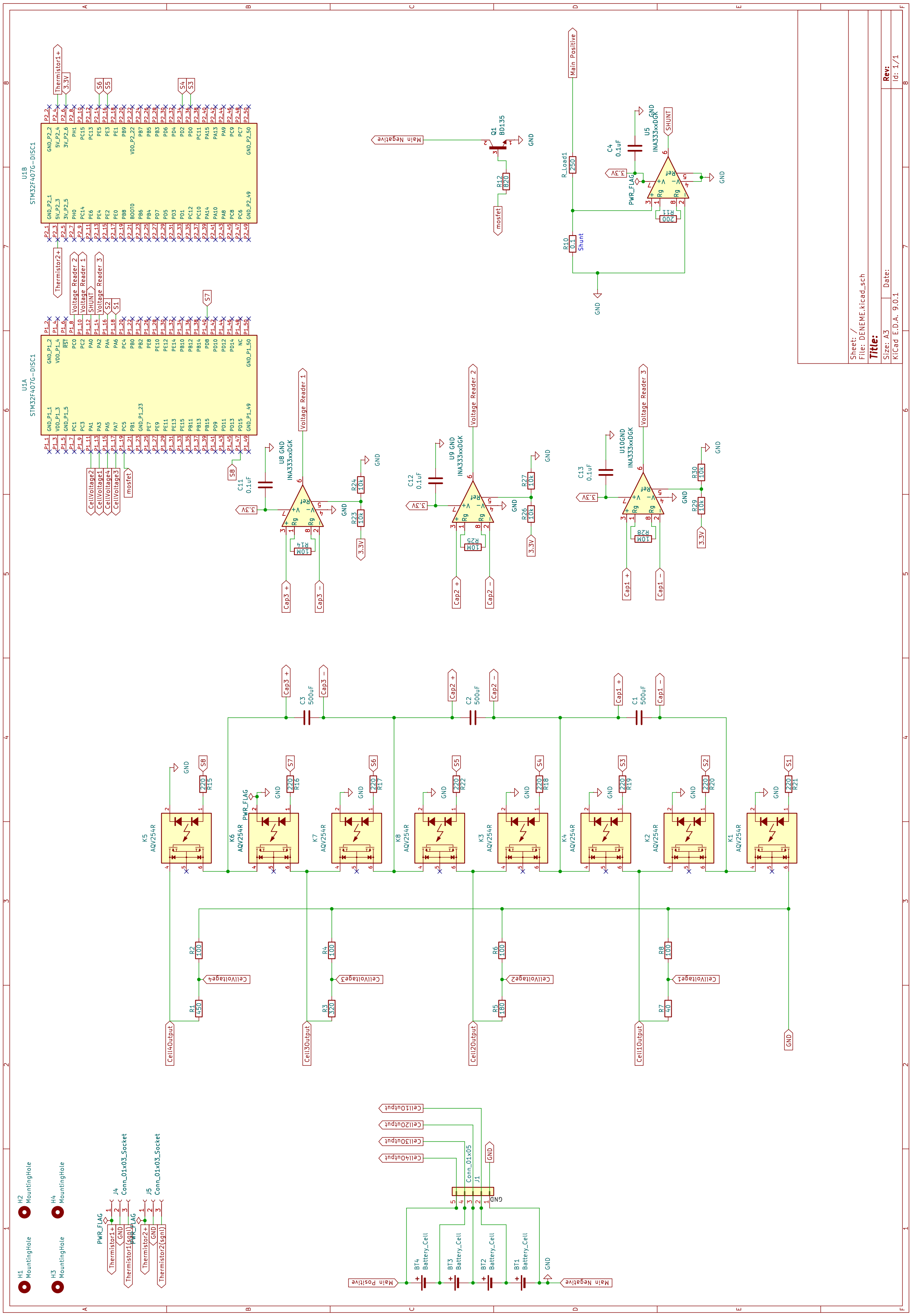

Figure 16: Whole schematic of PCB

All components used in this project, including microcontrollers, transistors, amplifiers, and passive elements, were either provided by the Özyeğin University. The PCB itself was manufactured within the university's electronics laboratory, ensuring full control over the design and fabrication process.

2.5 Schematic and Pcb

Figure 17 below shows the general view of the PCB designed for a BMS. This board is created to monitor and control a 4-cell li-ion battery pack connected in series. The design is based on the STM32F407G-DISC1 development board, which is used as the main controller. The PCB includes all necessary components for voltage monitoring, current measurement, temperature sensing, and battery protection. The PCB has a double-layer structure. A star-grounding method is used to keep analog and digital grounds separated, which improves measurement accuracy. The board also includes all necessary header connections for the STM32F407G-DISC1. It is suitable for both lab testing and real-world applications, allowing safe and efficient monitoring and control of the battery pack.

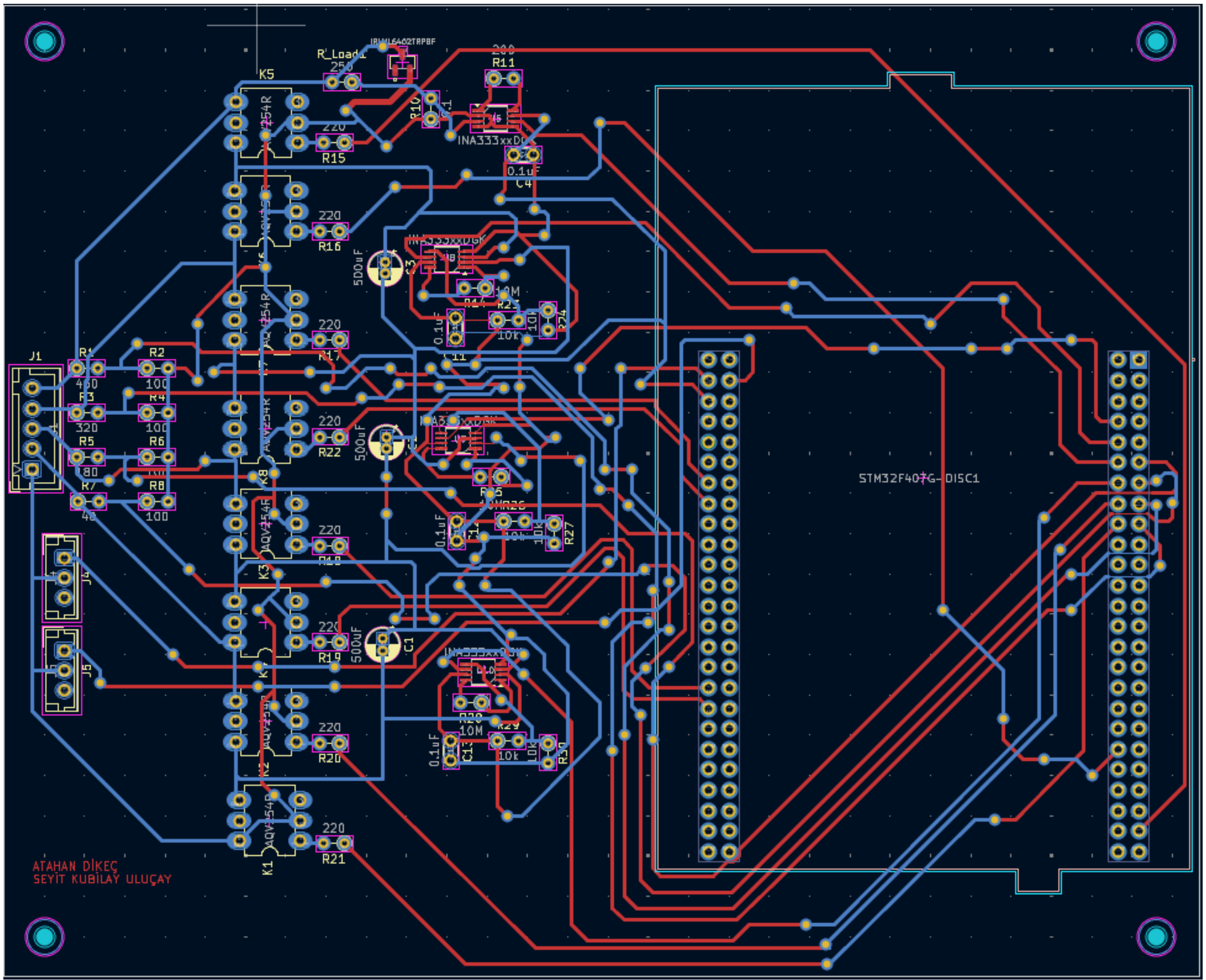

Figure 17: General look of PCB design.

For current measurement, the circuit uses an INA333 precision differential amplifieras shown in the figure 19 and 18 below. It reads the voltage drop across a shunt resistor and sends the amplified signal to the microcontroller's ADC (Analog-to-Digital Converter). The voltage of each battery cell is measured using voltage divider circuits, which reduce the cell voltage to a safe level for ADC inputs.

$$G = 1 + \left(\frac{100\,\mathrm{k}\Omega}{R_g}\right) \tag{1}$$

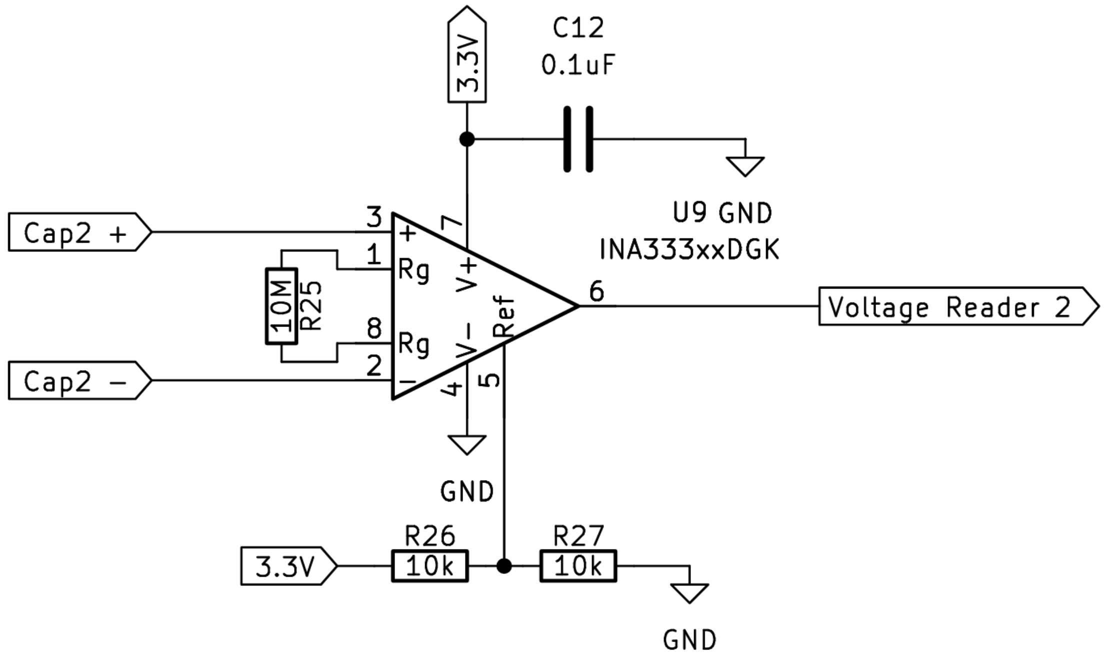

Figure 18: INA333 Diff. Amplifier connections (Rg = 10M).

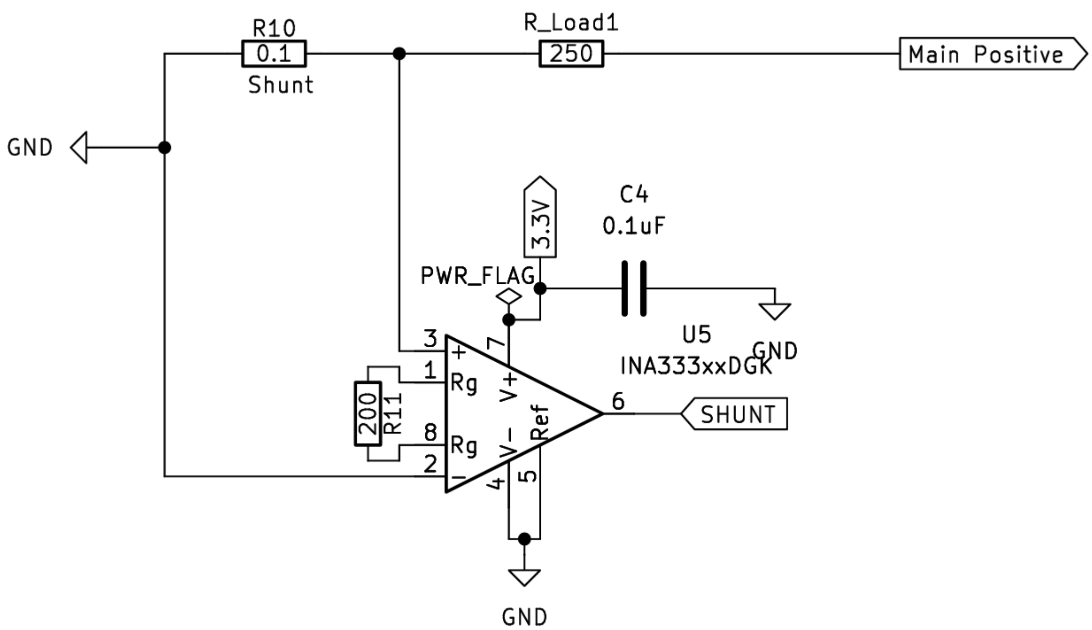

Figure 19: Amplification at the shunt resistor (Rg = 200).

NTC thermistors are used to monitor the temperature of each cell. In figure 20, the connections are demonstrated. These sensors help detect overheating by measuring resistance changes depending on temperature. The temperature values are also sent to the microcontroller via ADC pins.

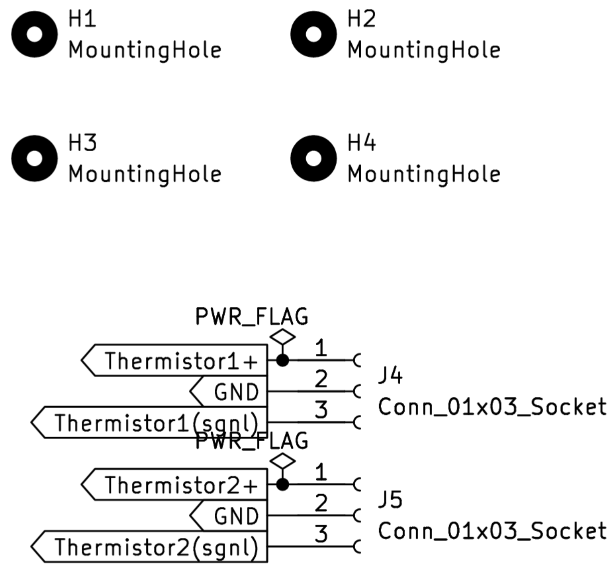

Figure 20: Mounting Holes and Thermistor connections at the schematic.

Two types of transistors are used in the circuit: AQV252G and BD135. AQV252G is a solid-state relay, which provides electrical isolation and allows safe switching in high-voltage sections. Every AQV252G ensures charging and discharging balance that works together with the algorithm of the system. It controls the voltage balance between capacitors and battery cells. Figure 6 shows the schematic of the system. In figure 2 the BD135 NPN transistor is used as a low-side switch to enhance system safety. It controls the connection between the system's main negative line and ground, allowing disconnection in fault conditions such as overcurrent or overheating. The base is driven through an 820 ohm resistor by a MOSFET output, enabling or disabling the ground path. This setup provides a reliable hardware-based method to isolate the battery when needed These transistors are controlled by the microcontroller's digital output pins.

Figure 21 shows the battery cell connections of the circuit.

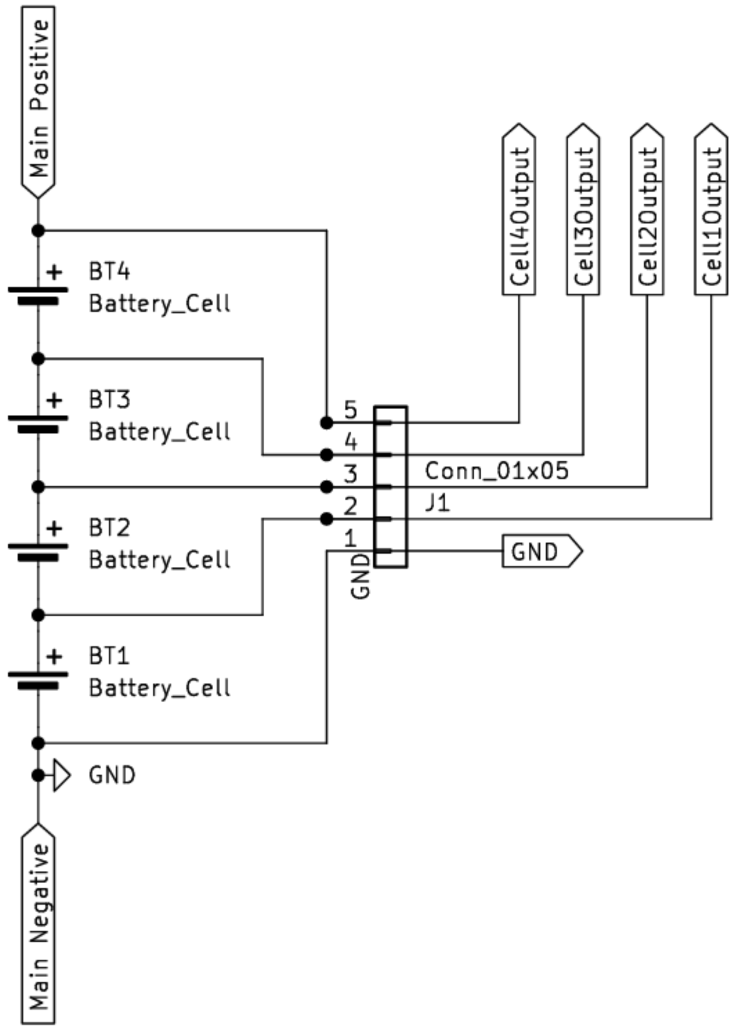

Figure 21: Battery cell connections.

After the cells are connected to PCB they directly connected to voltage divider with specific configurations as shown in the figure 22.

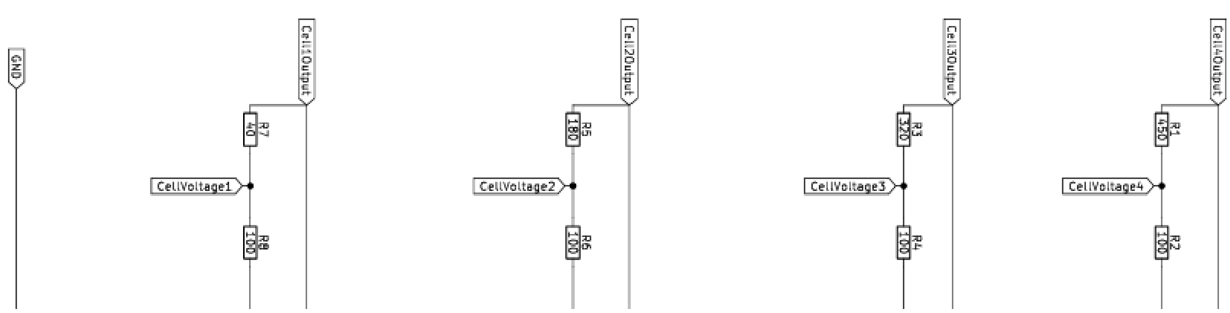

Figure 22: Voltage divider configurations

Figure 23 shows the 3D front look of the PCB in Kicad.

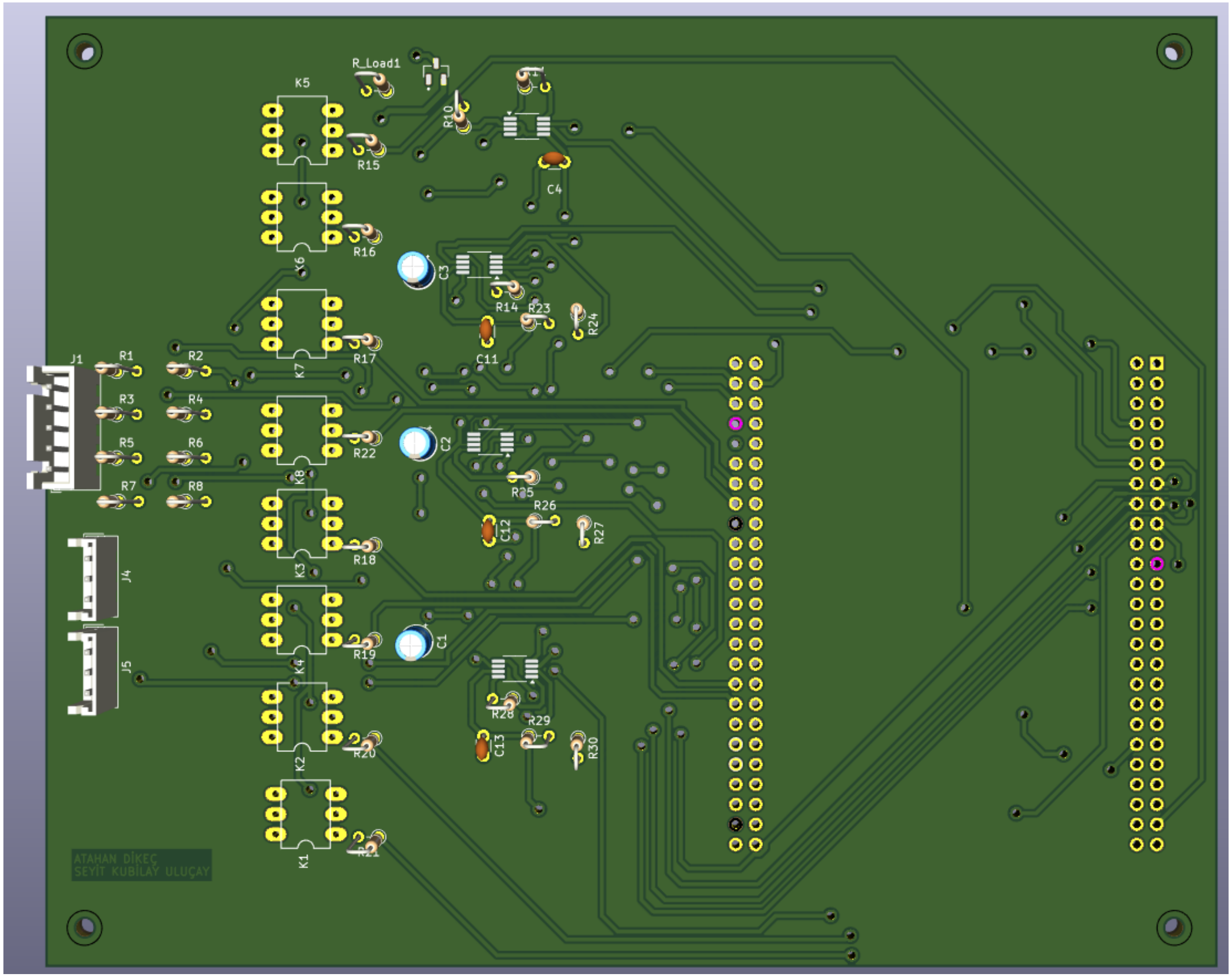

Figure 23: 3D front look of the PCB.

Figure 24 shows the 3D back look of the PCB in Kicad.

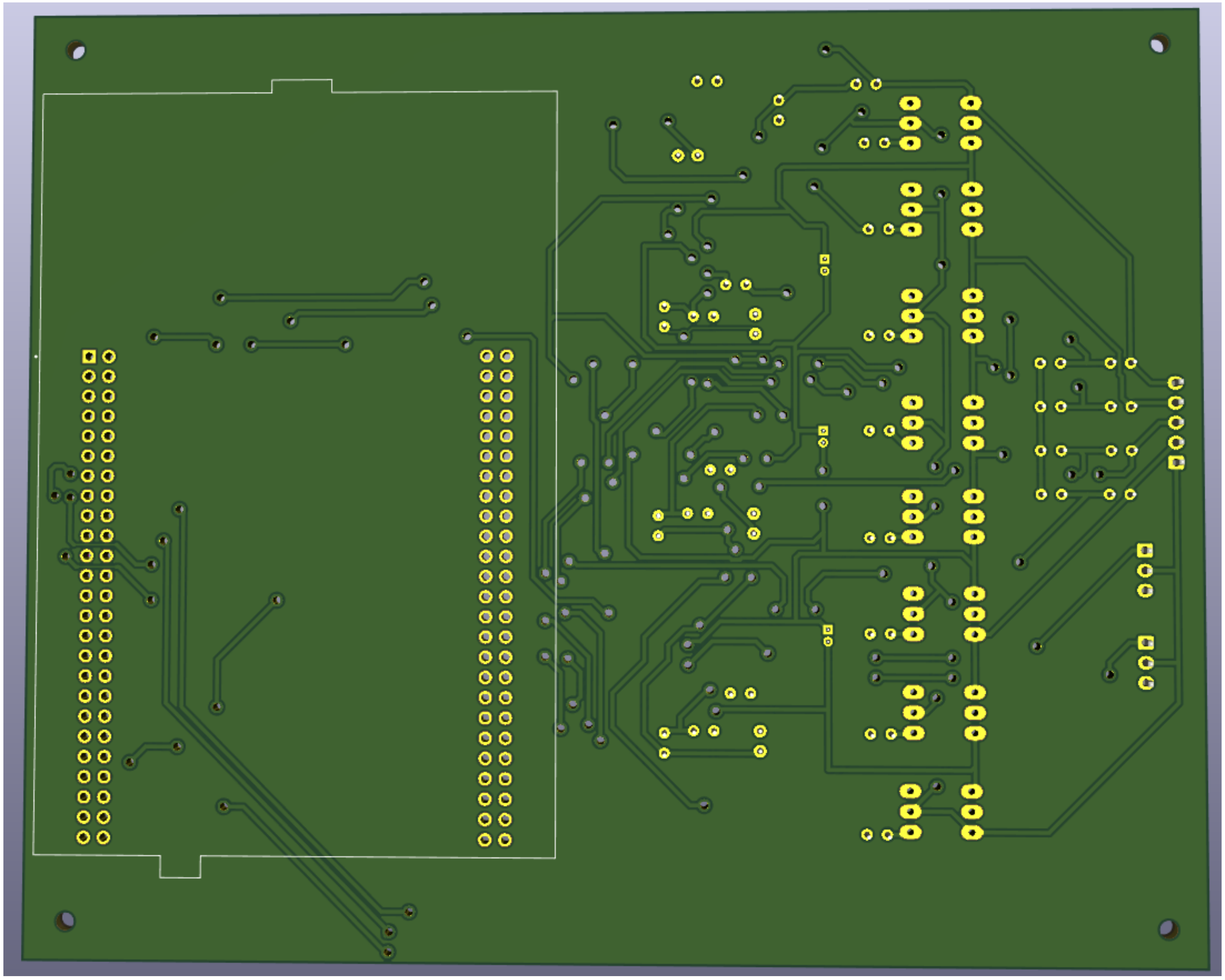

Figure 24: 3D back look of the PCB.

In the PCB design we had to use both sides effectively because the PCB was planned to be manufactured in the university laboratory, and because of that PCB sizing had to be below some constraints.

Also, the signals and power paths are located in different sides of the PCB, with that configuration, it is easier to follow up the signals or power pathways.

Figure 25 shows the real world look of the PCB without any component.

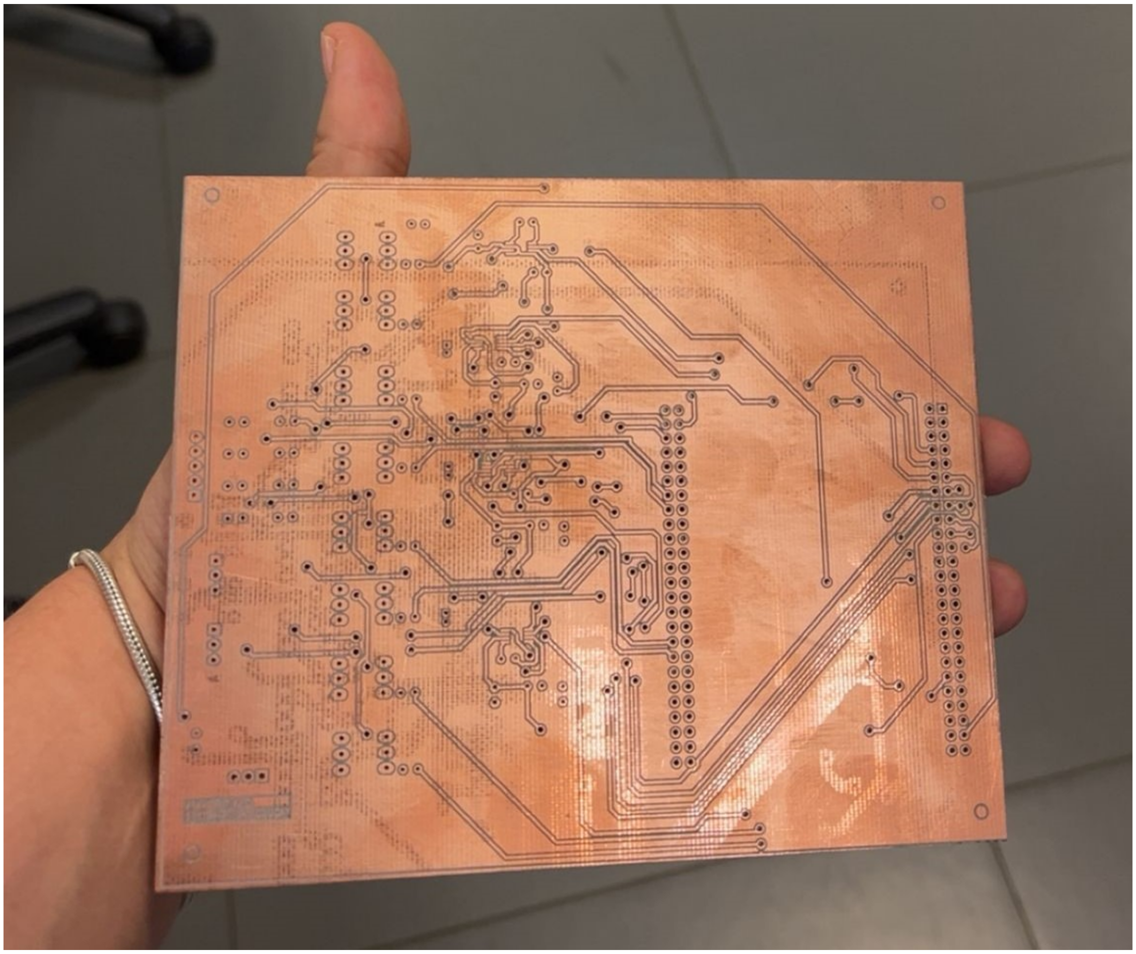

Figure 25: Printed PCB without any component.

Following the completion of the PCB manufacturing process, the board was visually inspected on both the front and back sides, as shown in Figures 26 and 27, respectively. The layout corresponds precisely to the finalized schematic design, ensuring accurate placement and routing of all components.

All component footprints, except for those used for resistors, were downloaded and implemented using the SnapMagic online library [6]. This ensured standard-compliant pad sizing, pin configuration accuracy, and mechanical compatibility across all major components, thereby reducing the likelihood of soldering and placement errors during assembly. The resistor footprints were created manually in accordance with the component dimensions available in the university's electronics laboratory.

All components used on the PCB, aside from those that were specifically ordered for the project, were supplied directly from the Özyeğin University Electronics Laboratory inventory. This ensured practical compatibility with the available stock and supported efficient and cost-effective development.

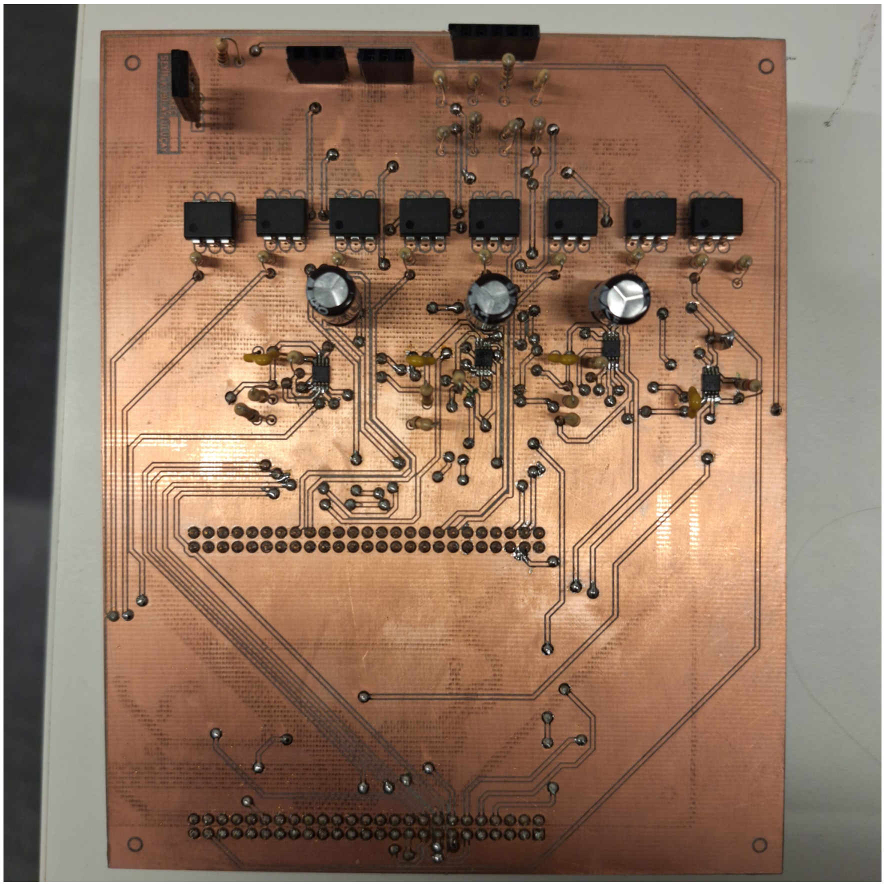

Figure 26: Front side of the finished PCB.

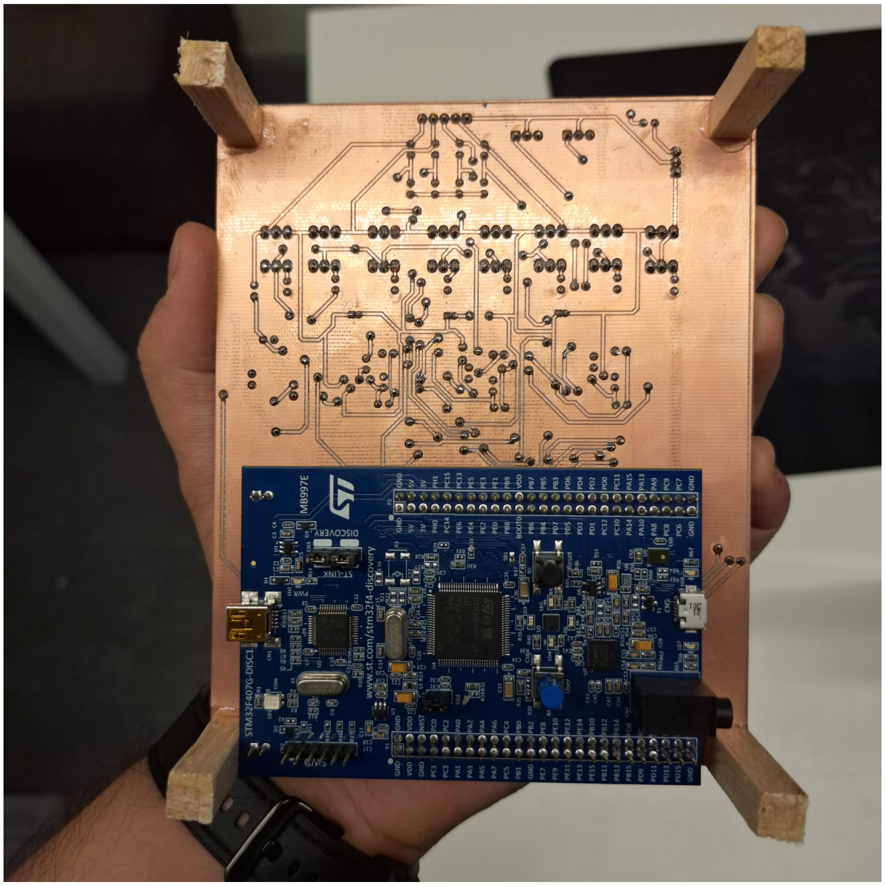

Figure 27: Back side of the finished PCB.

The algorithm was tested in Simulink and then adapted to the STM board. The code can be seen which is loaded onto the STM board in the appendix section.

3 RESULTS AND DISCUSSIONS

This section shows what we found from making and testing the BMS early model.

3.1 Results

A new Battery Management System was successfully created in MATLAB Simulink. This system uses capacitor-based active balancing, which works with a smart algorithm to balance the cells. An important feature of this BMS is that its parameters can be customized by the user in the required and desired way.
Key achievements and characteristics of the developed BMS are:

- **Real-Time Monitoring and Safety:** Temperature, voltage, and current are read in real-time. This allows for continuous safety checks and control of a dedicated safety switch transistor.
- **Custom Balancing Algorithm:** The most significant difference is the custom balancing control algorithm. This algorithm allows for the balancing of cells that may even have different sizes or capacities. This feature greatly increases the scalability of the system and the possibilities for customization.
- **Balancing Performance:** Successful cell balancing was observed during simulations in both charging and discharging states of the battery pack. The effectiveness of the algorithm in reducing voltage differences between cells was confirmed, as detailed in the analysis of simulation data (refer to Tables 1-4 and the accompanying analysis in Section 2.3.9).

3.2 Discussion

The developed BMS prototype highlights several advantages, mostly in terms of flexibility and adaptability due to its custom control algorithm. Offers a significant improvement over many existing solutions. Furthermore, the algorithm's capability to handle cells of potentially different characteristics enhances its practical applicability.

Real-time monitoring of critical parameters (voltage, current, temperature) is functional and provides the necessary data for both the balancing algorithm and the safety cut-off mechanisms. The successful balancing observed in simulation data for both charging and discharging scenarios validates the core logic of the custom algorithm.

However, it is important to note the current status of the system. While the core functionalities are established and have been validated through simulation, the system is not yet in a perfect or production-ready condition. The primary limitation at this stage is that the physical hardware setup has not been subjected to comprehensive long-term testing. These extended tests are crucial for evaluating the system's stability, reliability, and the durability of components under continuous operation. Such testing was not completed due to time constraints within the project period. Therefore, while simulation results are promising, the long-term real-world performance of the physical prototype remains to be fully characterized.

4 CONCLUSIONS AND FUTURE WORKS

A new Battery Management System featuring capacitor-based active cell balancing was successfully developed and validated through simulation. The core of this BMS is a smart, customizable control algorithm that effectively balances cells, allowing users to adjust parameters as needed.
The system incorporates real-time monitoring of essential parameters such as temperature, voltage, and current, which enables safety protocols, including the control of a safety switch transistor. The custom balancing algorithm is a significant innovation, offering the potential to balance cells of varying sizes or capacities, thereby enhancing system scalability and customization. Simulation results have demonstrated successful balancing performance in both charging and discharging states.

4.1 Future Works

While the current BMS prototype has achieved its primary design goals, several areas for future work have been identified to further enhance its performance and robustness:

- **Comprehensive Hardware Testing:** The most immediate future work involves conducting long-term tests on the physical hardware setup. This is important to evaluate its stability, reliability under continuous operation, and the durability of the components, which could not be fully completed due to project time constraints.
- **Optimization of Balancing Speed:** The current balancing algorithm focuses on one cell pair at a time. A significant improvement planned for future work is to modify the algorithm to allow for parallel balancing operations. It is anticipated that running the balancing logic for two or more cell pairs simultaneously could hugely increase the overall speed of cell equalization.
- **User Interface Development:** For enhanced usability, a simple user interface could be developed to display key battery parameters and allow for easier adjustment of customizable algorithm settings.
- **Expansion for More Cells:** While currently designed for up to four cells, the architectural principles could be extended and tested for battery packs with a larger number of cells.

Addressing these areas will contribute to the development of a more mature and field-ready BMS.

REFERENCES

[1] Trudyo, “Bd135 transistör npn to126,” Web Page, Jun. 2025, [Accessed: 2025-06-04]. Available: https://trudyo.com/magaza/bd135-transistor-npn-to126/.

[2] Mouser Electronics Turkey and Texas Instruments, “Ina333 düşük güç tüketimli, hassas enstrümantasyon amplifikatörleri,” Web Page, Jun. 2025, [Accessed: 2025-06-04]. Available: https://www.mouser.com.tr/new/texas-instruments/ti-ina333-instrumentation-amplifiers/.

[3] E-Komponent, “Aqv252g solid state dip6,” Web Page, Jun. 2025, [Accessed: 2025-06-04]. Available: https://www.e-komponent.com/solid-state-dpst-no-6-dip-aqv252g.

[4] RobotShop, “Temperature sensor tmp36,” Web Page, Jun. 2025, [Accessed: 2025-06-04]. Available: https://ca.robotshop.com/products/temperature-sensor-tmp36.

[5] MathWorks, “Balance battery cells with switched capacitor method,” Web Page, [Accessed: 2025-04-21]. [Online]. Available: https://www.mathworks.com/help/simscape-battery/ug/balance-battery-cells-with-switched-capacitor-method.html

[6] SnapEDA, “Snapeda - electronics design library (pcb footprints and schematic symbols),” Web Page, Jun. 2025, [Accessed: 2025-06-04]. Available: https://www.snapeda.com/.

You may find more about citing different sources (books, journals, articles, projects, dissertations, etc.) using the IEEE style from this link: IEEE Citation Style Guide (`ijssst.info`).

APPENDICES

Appendix-A Figure 28 is the MATLAB/Simulink simulation of the BMS with active balancing system.

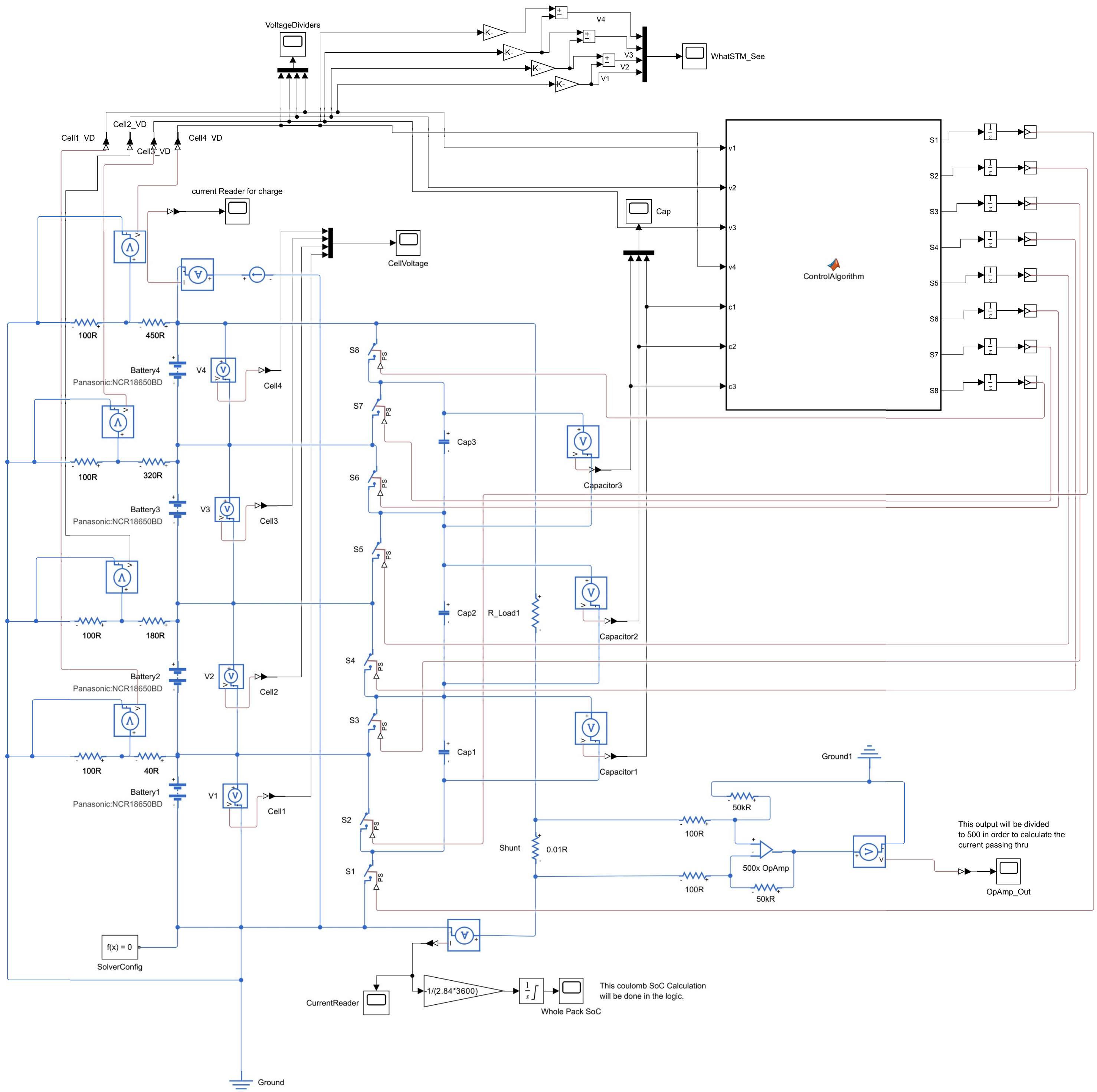

Figure 28: Simulation Overview

Appendix-B List 8 The code inside of the STM board.

```
/* USER CODE END Header */
/* Includes ------------------------------------------------------------------*/
#include "main.h"
// #include "usb_host.h"

/* Private includes ----------------------------------------------------------*/
/* USER CODE BEGIN Includes */
#include <stdio.h>
#include <math.h>
#include <stdarg.h>
/* USER CODE END Includes */

/* Private typedef -----------------------------------------------------------*/
/* USER CODE BEGIN PTD */
/* USER CODE END PTD */

/* Private define ------------------------------------------------------------*/
/* USER CODE BEGIN PD */
#define ADC_CHANNEL_COUNT 10
/* USER CODE END PD */

/* Private macro -------------------------------------------------------------*/
/* USER CODE BEGIN PM */
/* USER CODE END PM */

/* Private variables ---------------------------------------------------------*/
ADC_HandleTypeDef hadc1;
DMA_HandleTypeDef hdma_adc1;

// I2C_HandleTypeDef hi2c1;
/* USER CODE BEGIN PV */
// ADC DMA Buffer
uint16_t adc_values_raw[ADC_CHANNEL_COUNT];

// Reconstructed voltages and readings
float v_cell1_actual, v_cell2_actual, v_cell3_actual, v_cell4_actual;
float cap1_voltage, cap2_voltage, cap3_voltage; // Read but not used by current
```

```
    balancing algorithm
float pack_current;
float temp_sensor1_C, temp_sensor2_C;

// Control Algorithm Persistent State
static int balancingCap = 0;
static int pulsesRemaining = 0;
static int toggleState = 0;
static const int N_val = 10;

// Constants for the balancing algorithm
const float THRESHOLD_TO_PICK_NEW_PAIR = 0.010f; // 10mV
const float TARGET_DV_FOR_ACTIVE_PAIR  = 0.010f; // 10mV

// ADC & System Constants
const float ADC_VREF = 3.3f;    // Assuming VDDA is 3.3V
const float ADC_MAX_VALUE = 4095.0f; // For 12-bit ADC (2^12 - 1)

// Corrected Voltage Divider Gains (R2=100k for all, R1 values)
const float GAIN_SUM_C1          = 1.47f; // (47k+100k)/100k for VD1 (PA3)
const float GAIN_SUM_C1_C2       = 2.80f; // (180k+100k)/100k for VD2 (PA1)
const float GAIN_SUM_C1_C2_C3    = 4.30f; // (330k+100k)/100k for VD3 (PA7)
const float GAIN_SUM_C1_C2_C3_C4 = 5.70f; // (470k+100k)/100k for VD4 (PA5)

// Shunt related (PA0)
const float SHUNT_VOLTAGE_TO_CURRENT_DIVISOR = 50.0f;
// TMP36GZ Temperature Sensor Constants
const float TMP36_OFFSET_V = 0.5f;      // 0.5V at 0 C  for TMP36
const float TMP36_SCALE_FACTOR_V_PER_C = 0.01f; // 10mV per  C  for TMP36

// Safety Limits
const float MAX_CELL_VOLTAGE = 4.2f;
const float MIN_CELL_VOLTAGE = 2.8f;
const float MAX_TEMPERATURE = 55.0f; // Celsius
const float MIN_TEMPERATURE = -10.0f; // Celsius
// const float MAX_CURRENT_CHARGE = 20.0f; // Amps
// const float MAX_CURRENT_DISCHARGE = 50.0f; // Amps
uint8_t main_bjt_switch_on = 0; // 0 = OFF (Open Circuit), 1 = ON (Connected)
char printf_buffer[128]; // Buffer for SWV printf
```

```
/* USER CODE END PV */

/* Private function prototypes -----------------------------------------------*/
void SystemClock_Config(void);
static void MX_GPIO_Init(void);
static void MX_DMA_Init(void);
// static void MX_I2C1_Init(void);
static void MX_ADC1_Init(void);
// void MX_USB_HOST_Process(void);
/* USER CODE BEGIN PFP */
void ControlAlgorithm_C_Revised(float v1_div, float v2_div, float v3_div, float v4_div,
                                uint8_t* S_out);
float ConvertTMP36ADCtoTemperature(uint16_t adc_raw_sensor);
void SWV_Printf(const char *fmt, ...);
/* USER CODE END PFP */

/* Private user code ---------------------------------------------------------*/
/* USER CODE BEGIN 0 */
/* USER CODE END 0 */

/**
  * @brief  The application entry point.
  * @retval int
  */
int main(void)
{
  /* USER CODE BEGIN 1 */
  /* USER CODE END 1 */

  /* MCU Configuration--------------------------------------------------------*/
  HAL_Init();
  /* USER CODE BEGIN Init */
  /* USER CODE END Init */
  SystemClock_Config();
  /* USER CODE BEGIN SysInit */
  /* USER CODE END SysInit */

  /* Initialize all configured peripherals */
  MX_GPIO_Init();
```

```c
MX_DMA_Init();
// MX_I2C1_Init();
// MX_USB_HOST_Init();
MX_ADC1_Init();
/* USER CODE BEGIN 2 */
// Start ADC with DMA
if (HAL_ADC_Start_DMA(&hadc1, (uint32_t*)adc_values_raw, ADC_CHANNEL_COUNT) != HAL_OK) {
    Error_Handler(); // ADC DMA Start Error
}

// Initial state: Main BJT Switch OFF (safer) until first check
HAL_GPIO_WritePin(Mosfet_GPIO_Port, Mosfet_Pin, GPIO_PIN_RESET); // PC5 - Main BJT Switch OFF
main_bjt_switch_on = 0;
/* USER CODE END 2 */

/* Infinite loop */
/* USER CODE BEGIN WHILE */
while (1)
{
/* USER CODE END WHILE */
// MX_USB_HOST_Process();

/* USER CODE BEGIN 3 */
      // ADC Value Order (Rank 1 to 10 -> index 0 to 9)
      // Based on your User Labels and typical INx mapping:
      // adc_values_raw[0]: shunt (PA0 -> IN0)
      // adc_values_raw[1]: VD2   (PA1 -> IN1 -> for Cell1+Cell2 sum)
      // adc_values_raw[2]: C3_Read (PA2 -> IN2 -> Cap3 voltage)
      // adc_values_raw[3]: VD1   (PA3 -> IN3 -> for Cell1 sum)
      // adc_values_raw[4]: VD4   (PA5 -> IN5 -> for Cell1+2+3+4 sum)
      // adc_values_raw[5]: VD3   (PA7 -> IN7 -> for Cell1+2+3 sum)
      // adc_values_raw[6]: C2_Read (PC0 -> IN10 -> Cap2 voltage)
      // adc_values_raw[7]: TMP1  (PC1 -> IN11 -> Temp Sensor 1)
      // adc_values_raw[8]: C1_Read (PC2 -> IN12 -> Cap1 voltage)
      // adc_values_raw[9]: TMP2  (PC4 -> IN14 -> Temp Sensor 2)

  // Convert raw ADC values from DMA buffer to voltages
```

```c
    float shunt_adc_voltage   = (adc_values_raw[0] / ADC_MAX_VALUE) * ADC_VREF;
    float v2_raw_div_out      = (adc_values_raw[1] / ADC_MAX_VALUE) * ADC_VREF; // Input for Cell1+Cell2 sum
    cap3_voltage              = (adc_values_raw[2] / ADC_MAX_VALUE) * ADC_VREF; // Cap3 voltage
    float v1_raw_div_out      = (adc_values_raw[3] / ADC_MAX_VALUE) * ADC_VREF; // Input for Cell1 sum
    float v4_raw_div_out      = (adc_values_raw[4] / ADC_MAX_VALUE) * ADC_VREF; // Input for Cell1+2+3+4 sum
    float v3_raw_div_out      = (adc_values_raw[5] / ADC_MAX_VALUE) * ADC_VREF; // Input for Cell1+2+3 sum
    cap2_voltage              = (adc_values_raw[6] / ADC_MAX_VALUE) * ADC_VREF; // Cap2 voltage
    uint16_t tmp1_adc_raw     = adc_values_raw[7];                             // TMP1 raw ADC
    cap1_voltage              = (adc_values_raw[8] / ADC_MAX_VALUE) * ADC_VREF; // Cap1 voltage
    uint16_t tmp2_adc_raw     = adc_values_raw[9];                             // TMP2 raw ADC

    // Reconstruct ACTUAL individual cell voltages for safety checks
    float sum_c1_calc          = GAIN_SUM_C1 * v1_raw_div_out;
    float sum_c1_c2_calc       = GAIN_SUM_C1_C2 * v2_raw_div_out;
    float sum_c1_c2_c3_calc    = GAIN_SUM_C1_C2_C3 * v3_raw_div_out;
    float sum_c1_c2_c3_c4_calc = GAIN_SUM_C1_C2_C3_C4 * v4_raw_div_out;

    v_cell1_actual = sum_c1_calc;
    v_cell2_actual = sum_c1_c2_calc - sum_c1_calc;
    v_cell3_actual = sum_c1_c2_c3_calc - sum_c1_c2_calc;
    v_cell4_actual = sum_c1_c2_c3_c4_calc - sum_c1_c2_c3_calc;

    // Calculate pack current
    pack_current = shunt_adc_voltage / SHUNT_VOLTAGE_TO_CURRENT_DIVISOR;

    // Convert TMP36 readings to temperature
    temp_sensor1_C = ConvertTMP36ADCtoTemperature(tmp1_adc_raw);
    temp_sensor2_C = ConvertTMP36ADCtoTemperature(tmp2_adc_raw);

    // --- Main BJT Switch (PC5 - User Label "Mosfet") Safety Control ---
```

```
    uint8_t open_main_switch_flag = 0;
    if (v_cell1_actual > MAX_CELL_VOLTAGE || v_cell2_actual > MAX_CELL_VOLTAGE ||
        v_cell3_actual > MAX_CELL_VOLTAGE || v_cell4_actual > MAX_CELL_VOLTAGE ||
        v_cell1_actual < MIN_CELL_VOLTAGE || v_cell2_actual < MIN_CELL_VOLTAGE ||
        v_cell3_actual < MIN_CELL_VOLTAGE || v_cell4_actual < MIN_CELL_VOLTAGE ) {
        open_main_switch_flag = 1; // Cell voltage out of safe range
    }
    if (temp_sensor1_C > MAX_TEMPERATURE || temp_sensor2_C > MAX_TEMPERATURE ||
        temp_sensor1_C < MIN_TEMPERATURE || temp_sensor2_C < MIN_TEMPERATURE) {
        open_main_switch_flag = 1; // Temperature out of safe range
    }
    // Add other safety checks here but not quite possible (maybe pack_current limits etc. idk) and set open_main_switch_flag = 1 if fault

    if (open_main_switch_flag) {
        if (main_bjt_switch_on) { // If it was ON, turn it OFF
            HAL_GPIO_WritePin(Mosfet_GPIO_Port, Mosfet_Pin, GPIO_PIN_RESET);
            main_bjt_switch_on = 0;
        }
    } else { // No fault condition
        if (!main_bjt_switch_on) { // If it was OFF, turn it ON
            HAL_GPIO_WritePin(Mosfet_GPIO_Port, Mosfet_Pin, GPIO_PIN_SET);
            main_bjt_switch_on = 1;
        }
    }

    // --- Balancing Logic ---
    uint8_t S_switches_cmd[8] = {0}; // Initialize all switch commands to OFF
    if (main_bjt_switch_on) { // Only perform balancing if the main BJT switch is ON (pack is connected)
        // Call balancing algorithm with RAW divider outputs (v1_raw_div_out, etc.)
        ControlAlgorithm_C_Revised(v1_raw_div_out, v2_raw_div_out, v3_raw_div_out, v4_raw_div_out,
                                   S_switches_cmd);
    } else {
        // If main BJT switch is OFF, ensure balancing algorithm state is reset
        balancingCap = 0;
        pulsesRemaining = 0;
        toggleState = 0;
```

```
      // S_switches_cmd remains all zeros (all balancing switches OFF)
    }

    // Apply switch commands (S1-S8) using User Labels from main.h
    HAL_GPIO_WritePin(S1_GPIO_Port, S1_Pin, (GPIO_PinState)S_switches_cmd[0]);
    HAL_GPIO_WritePin(S2_GPIO_Port, S2_Pin, (GPIO_PinState)S_switches_cmd[1]);
    HAL_GPIO_WritePin(S3_GPIO_Port, S3_Pin, (GPIO_PinState)S_switches_cmd[2]);
    HAL_GPIO_WritePin(S4_GPIO_Port, S4_Pin, (GPIO_PinState)S_switches_cmd[3]);
    HAL_GPIO_WritePin(S5_GPIO_Port, S5_Pin, (GPIO_PinState)S_switches_cmd[4]);
    HAL_GPIO_WritePin(S6_GPIO_Port, S6_Pin, (GPIO_PinState)S_switches_cmd[5]);
    HAL_GPIO_WritePin(S7_GPIO_Port, S7_Pin, (GPIO_PinState)S_switches_cmd[6]);
    HAL_GPIO_WritePin(S8_GPIO_Port, S8_Pin, (GPIO_PinState)S_switches_cmd[7]);

    // SWV Debug Print (for debuggggg)
    static uint32_t last_print_time = 0;
    if (HAL_GetTick() - last_print_time >= 1000) { // Print every 1 second
      SWV_Printf("V:%.2f,%.2f,%.2f,%.2f C:%.2f T:%.1f,%.1f BJT:%d BAL:%d PLS:%d\r\n",
                 v_cell1_actual, v_cell2_actual, v_cell3_actual, v_cell4_actual,
                 pack_current, temp_sensor1_C, temp_sensor2_C,
                 main_bjt_switch_on, balancingCap, pulsesRemaining);
      last_print_time = HAL_GetTick();
    }

    HAL_Delay(1); // Control loop delay (1ms)
  }
  /* USER CODE END 3 */
}

/**
  * @brief System Clock Configuration
  * @retval None
  */
void SystemClock_Config(void)
{
  RCC_OscInitTypeDef RCC_OscInitStruct = {0};
  RCC_ClkInitTypeDef RCC_ClkInitStruct = {0};

  /** Configure the main internal regulator output voltage
  */
```

```c
  __HAL_RCC_PWR_CLK_ENABLE();
  __HAL_PWR_VOLTAGESCALING_CONFIG(PWR_REGULATOR_VOLTAGE_SCALE1);

  /** Initializes the RCC Oscillators according to the specified parameters
  * in the RCC_OscInitTypeDef structure.
  */
  RCC_OscInitStruct.OscillatorType = RCC_OSCILLATORTYPE_HSE;
  RCC_OscInitStruct.HSEState = RCC_HSE_BYPASS; // Or RCC_HSE_ON if using a crystal directly
  RCC_OscInitStruct.PLL.PLLState = RCC_PLL_ON;
  RCC_OscInitStruct.PLL.PLLSource = RCC_PLLSOURCE_HSE;
  RCC_OscInitStruct.PLL.PLLM = 8; // Value for 8MHz HSE on STLink MCO or external 8MHz crystal
  RCC_OscInitStruct.PLL.PLLN = 336;
  RCC_OscInitStruct.PLL.PLLP = RCC_PLLP_DIV2;
  RCC_OscInitStruct.PLL.PLLQ = 7; // For USB OTG FS
  if (HAL_RCC_OscConfig(&RCC_OscInitStruct) != HAL_OK)
  {
    Error_Handler();
  }

  /** Initializes the CPU, AHB and APB buses clocks
  */
  RCC_ClkInitStruct.ClockType = RCC_CLOCKTYPE_HCLK|RCC_CLOCKTYPE_SYSCLK
                              |RCC_CLOCKTYPE_PCLK1|RCC_CLOCKTYPE_PCLK2;
  RCC_ClkInitStruct.SYSCLKSource = RCC_SYSCLKSOURCE_PLLCLK;
  RCC_ClkInitStruct.AHBCLKDivider = RCC_SYSCLK_DIV1;
  RCC_ClkInitStruct.APB1CLKDivider = RCC_HCLK_DIV4;
  RCC_ClkInitStruct.APB2CLKDivider = RCC_HCLK_DIV2;

  if (HAL_RCC_ClockConfig(&RCC_ClkInitStruct, FLASH_LATENCY_5) != HAL_OK)
  {
    Error_Handler();
  }
}

/**
  * @brief ADC1 Initialization Function
  * @param None
```

```
  * @retval None
  */
static void MX_ADC1_Init(void)
{
  /* USER CODE BEGIN ADC1_Init 0 */
  /* USER CODE END ADC1_Init 0 */
  ADC_ChannelConfTypeDef sConfig = {0};
  /* USER CODE BEGIN ADC1_Init 1 */
  /* USER CODE END ADC1_Init 1 */


  hadc1.Instance = ADC1;
  hadc1.Init.ClockPrescaler = ADC_CLOCK_SYNC_PCLK_DIV4;
  hadc1.Init.Resolution = ADC_RESOLUTION_12B;
  hadc1.Init.ScanConvMode = ENABLE;
  hadc1.Init.ContinuousConvMode = ENABLE;
  hadc1.Init.DiscontinuousConvMode = DISABLE;
  hadc1.Init.ExternalTrigConvEdge = ADC_EXTERNALTRIGCONVEDGE_NONE;
  hadc1.Init.ExternalTrigConv = ADC_SOFTWARE_START;
  hadc1.Init.DataAlign = ADC_DATAALIGN_RIGHT;
  hadc1.Init.NbrOfConversion = ADC_CHANNEL_COUNT; // Use the define
  hadc1.Init.DMAContinuousRequests = ENABLE;
  hadc1.Init.EOCSelection = ADC_EOC_SEQ_CONV;
  if (HAL_ADC_Init(&hadc1) != HAL_OK)
  {
    Error_Handler();
  }


  sConfig.SamplingTime = ADC_SAMPLETIME_28CYCLES;

  sConfig.Channel = ADC_CHANNEL_0;  sConfig.Rank = 1;  // shunt (PA0)
  if (HAL_ADC_ConfigChannel(&hadc1, &sConfig) != HAL_OK) { Error_Handler(); }
  sConfig.Channel = ADC_CHANNEL_1;  sConfig.Rank = 2;  // VD2 (PA1)
  if (HAL_ADC_ConfigChannel(&hadc1, &sConfig) != HAL_OK) { Error_Handler(); }
  sConfig.Channel = ADC_CHANNEL_2;  sConfig.Rank = 3;  // C3_Read (PA2)
  if (HAL_ADC_ConfigChannel(&hadc1, &sConfig) != HAL_OK) { Error_Handler(); }
  sConfig.Channel = ADC_CHANNEL_3;  sConfig.Rank = 4;  // VD1 (PA3)
  if (HAL_ADC_ConfigChannel(&hadc1, &sConfig) != HAL_OK) { Error_Handler(); }
```

```
  sConfig.Channel = ADC_CHANNEL_5;  sConfig.Rank = 5;  // VD4 (PA5)
  if (HAL_ADC_ConfigChannel(&hadc1, &sConfig) != HAL_OK) { Error_Handler(); }
  sConfig.Channel = ADC_CHANNEL_7;  sConfig.Rank = 6;  // VD3 (PA7)
  if (HAL_ADC_ConfigChannel(&hadc1, &sConfig) != HAL_OK) { Error_Handler(); }
  sConfig.Channel = ADC_CHANNEL_10; sConfig.Rank = 7;  // C2_Read (PC0)
  if (HAL_ADC_ConfigChannel(&hadc1, &sConfig) != HAL_OK) { Error_Handler(); }
  sConfig.Channel = ADC_CHANNEL_11; sConfig.Rank = 8;  // TMP1 (PC1)
  if (HAL_ADC_ConfigChannel(&hadc1, &sConfig) != HAL_OK) { Error_Handler(); }
  sConfig.Channel = ADC_CHANNEL_12; sConfig.Rank = 9;  // C1_Read (PC2)
  if (HAL_ADC_ConfigChannel(&hadc1, &sConfig) != HAL_OK) { Error_Handler(); }
  sConfig.Channel = ADC_CHANNEL_14; sConfig.Rank = 10; // TMP2 (PC4)
  if (HAL_ADC_ConfigChannel(&hadc1, &sConfig) != HAL_OK) { Error_Handler(); }
  /* USER CODE BEGIN ADC1_Init 2 */
  /* USER CODE END ADC1_Init 2 */
}

/**
  * @brief I2C1 Initialization Function
  * @param None
  * @retval None
  */
// static void MX_I2C1_Init(void)
// {
//   /* USER CODE BEGIN I2C1_Init 0 */
//   /* USER CODE END I2C1_Init 0 */
//   /* USER CODE BEGIN I2C1_Init 1 */
//   /* USER CODE END I2C1_Init 1 */
//   // hi2c1.Instance = I2C1;
//   /* USER CODE BEGIN I2C1_Init 2 */
//   /* USER CODE END I2C1_Init 2 */
// }

/**
  * Enable DMA controller clock
  */
static void MX_DMA_Init(void)
{
  /* DMA controller clock enable */
  __HAL_RCC_DMA2_CLK_ENABLE();
```

```c

  /* DMA interrupt init */
  /* DMA2_Stream0_IRQn interrupt configuration */
  HAL_NVIC_SetPriority(DMA2_Stream0_IRQn, 0, 0);
  HAL_NVIC_EnableIRQ(DMA2_Stream0_IRQn);
}

/**
  * @brief GPIO Initialization Function
  * @param None
  * @retval None
  */
static void MX_GPIO_Init(void)
{
  GPIO_InitTypeDef GPIO_InitStruct = {0};
  /* USER CODE BEGIN MX_GPIO_Init_1 */
  /* USER CODE END MX_GPIO_Init_1 */

  /* GPIO Ports Clock Enable */
  __HAL_RCC_GPIOE_CLK_ENABLE();
  __HAL_RCC_GPIOC_CLK_ENABLE();
  __HAL_RCC_GPIOH_CLK_ENABLE();
  __HAL_RCC_GPIOA_CLK_ENABLE();
  __HAL_RCC_GPIOB_CLK_ENABLE();
  __HAL_RCC_GPIOD_CLK_ENABLE();

  /*Configure GPIO pin Output Level for all BMS outputs to LOW */
  HAL_GPIO_WritePin(S1_GPIO_Port, S1_Pin, GPIO_PIN_RESET);       // PA6
  HAL_GPIO_WritePin(S2_GPIO_Port, S2_Pin, GPIO_PIN_RESET);       // PA4
  HAL_GPIO_WritePin(S3_GPIO_Port, S3_Pin, GPIO_PIN_RESET);       // PD0
  HAL_GPIO_WritePin(S4_GPIO_Port, S4_Pin, GPIO_PIN_RESET);       // PD2
  HAL_GPIO_WritePin(S5_GPIO_Port, S5_Pin, GPIO_PIN_RESET);       // PE3
  HAL_GPIO_WritePin(S6_GPIO_Port, S6_Pin, GPIO_PIN_RESET);       // PE5
  HAL_GPIO_WritePin(S7_GPIO_Port, S7_Pin, GPIO_PIN_RESET);       // PD8
  HAL_GPIO_WritePin(S8_GPIO_Port, S8_Pin, GPIO_PIN_RESET);       // PD15
  HAL_GPIO_WritePin(Mosfet_GPIO_Port, Mosfet_Pin, GPIO_PIN_RESET); // PC5

  /*Configure Output Pins */
  GPIO_InitStruct.Mode = GPIO_MODE_OUTPUT_PP;
```

```
  GPIO_InitStruct.Pull = GPIO_NOPULL;
  GPIO_InitStruct.Speed = GPIO_SPEED_FREQ_LOW;

  GPIO_InitStruct.Pin = S1_Pin; HAL_GPIO_Init(S1_GPIO_Port, &GPIO_InitStruct);
  GPIO_InitStruct.Pin = S2_Pin; HAL_GPIO_Init(S2_GPIO_Port, &GPIO_InitStruct);
  GPIO_InitStruct.Pin = S3_Pin; HAL_GPIO_Init(S3_GPIO_Port, &GPIO_InitStruct);
  GPIO_InitStruct.Pin = S4_Pin; HAL_GPIO_Init(S4_GPIO_Port, &GPIO_InitStruct);
  GPIO_InitStruct.Pin = S5_Pin; HAL_GPIO_Init(S5_GPIO_Port, &GPIO_InitStruct);
  GPIO_InitStruct.Pin = S6_Pin; HAL_GPIO_Init(S6_GPIO_Port, &GPIO_InitStruct);
  GPIO_InitStruct.Pin = S7_Pin; HAL_GPIO_Init(S7_GPIO_Port, &GPIO_InitStruct);
  GPIO_InitStruct.Pin = S8_Pin; HAL_GPIO_Init(S8_GPIO_Port, &GPIO_InitStruct);
  GPIO_InitStruct.Pin = Mosfet_Pin; HAL_GPIO_Init(Mosfet_GPIO_Port, &GPIO_InitStruct);

  // GPIO_InitStruct.Pin = BOOT1_Pin;
  // GPIO_InitStruct.Mode = GPIO_MODE_INPUT;
  // GPIO_InitStruct.Pull = GPIO_NOPULL;
  // HAL_GPIO_Init(BOOT1_GPIO_Port, &GPIO_InitStruct);

  /* USER CODE BEGIN MX_GPIO_Init_2 */
  /* USER CODE END MX_GPIO_Init_2 */
}

/* USER CODE BEGIN 4 */
void ControlAlgorithm_C_Revised(float v1_div, float v2_div, float v3_div, float v4_div,
                                uint8_t* S_out) {
    // --- 1. Reconstruct the *actual* cell voltages ---
    float sum1_eff    = GAIN_SUM_C1 * v1_div;
    float sum12_eff   = GAIN_SUM_C1_C2 * v2_div;
    float sum123_eff  = GAIN_SUM_C1_C2_C3 * v3_div;
    float sum1234_eff = GAIN_SUM_C1_C2_C3_C4 * v4_div;

    float Vcell1 = sum1_eff;
    float Vcell2 = sum12_eff - sum1_eff;
    float Vcell3 = sum123_eff - sum12_eff;
    float Vcell4 = sum1234_eff - sum123_eff;

    // --- 2. Compute adjacent cell differences ---
    float dV12 = Vcell2 - Vcell1;
    float dV23 = Vcell3 - Vcell2;
```

```
    float dV34 = Vcell4 - Vcell3;
    float abs_dV12 = fabsf(dV12);
    float abs_dV23 = fabsf(dV23);
    float abs_dV34 = fabsf(dV34);

    // --- 3. Decision Logic ---
    if (balancingCap != 0) {
        float current_active_dV_abs = 0.0f;
        if (balancingCap == 1) current_active_dV_abs = abs_dV12;
        else if (balancingCap == 2) current_active_dV_abs = abs_dV23;
        else if (balancingCap == 3) current_active_dV_abs = abs_dV34;

        if (current_active_dV_abs <= TARGET_DV_FOR_ACTIVE_PAIR) {
            balancingCap = 0; pulsesRemaining = 0; toggleState = 0;
        } else {
            if (pulsesRemaining == 0) {
                pulsesRemaining = 2 * N_val;
            }
        }
    }

    if (balancingCap == 0) {
        float abs_dVs[] = {abs_dV12, abs_dV23, abs_dV34}; // Array of absolute differences
        float maxDV_overall = 0.0f; // Max difference found so far
        int new_balancing_cap_idx = 0; // Index of the cap to balance (1, 2, or 3)

        // Find the pair with the largest absolute difference that is also above the threshold
        for(int i=0; i<3; ++i) {
            if(abs_dVs[i] > maxDV_overall && abs_dVs[i] > THRESHOLD_TO_PICK_NEW_PAIR) {
                maxDV_overall = abs_dVs[i];
                new_balancing_cap_idx = i + 1; // +1 because array is 0-indexed, balancingCap is 1,2,3
            }
        }

        if (new_balancing_cap_idx != 0) { // If a valid cap was found
            balancingCap    = new_balancing_cap_idx;
```

```
            pulsesRemaining = 2 * N_val;
            toggleState     = 0;
        } else { // No pair needs balancing
            balancingCap = 0; pulsesRemaining = 0; toggleState = 0;
        }
    }

    // --- 4. If a cap is active, execute one half stroke ---
    for (int i = 0; i < 8; ++i) S_out[i] = 0; // Initialize all switch commands to OFF

    if (balancingCap != 0 && pulsesRemaining > 0) {
        if (balancingCap == 1) {
            if (dV12 > 0) { if (toggleState == 0) { S_out[1] = 1; S_out[3] = 1; } else { S_out[0] = 1; S_out[2] = 1; } }
            else { if (toggleState == 0) { S_out[0] = 1; S_out[2] = 1; } else { S_out[1] = 1; S_out[3] = 1; } }
        } else if (balancingCap == 2) {
            if (dV23 > 0) { if (toggleState == 0) { S_out[3] = 1; S_out[5] = 1; } else { S_out[2] = 1; S_out[4] = 1; } }
            else { if (toggleState == 0) { S_out[2] = 1; S_out[4] = 1; } else { S_out[3] = 1; S_out[5] = 1; } }
        } else if (balancingCap == 3) {
            if (dV34 > 0) { if (toggleState == 0) { S_out[5] = 1; S_out[7] = 1; } else { S_out[4] = 1; S_out[6] = 1; } }
            else { if (toggleState == 0) { S_out[4] = 1; S_out[6] = 1; } else { S_out[5] = 1; S_out[7] = 1; } }
        }
        pulsesRemaining--;
        toggleState = 1 - toggleState;
    }

    // --- 5. Enforce noshort rules ---
    if (S_out[7] && S_out[5]) { S_out[6] = 0; S_out[4] = 0; }
    if (S_out[6] && S_out[4]) { S_out[7] = 0; S_out[5] = 0; S_out[3] = 0; S_out[1] = 0; }
    if (S_out[5] && S_out[3]) { S_out[7]=0; S_out[6]=0; S_out[4]=0; S_out[2]=0; S_out[1]=0; S_out[0]=0; }
    if (S_out[4] && S_out[2]) { S_out[7]=0; S_out[6]=0; S_out[5]=0; S_out[3]=0; S_out[1]=0; S_out[0]=0; }
```

```c
    if (S_out[3] && S_out[1]) { S_out[6]=0; S_out[4]=0; S_out[2]=0; S_out[0]=0; }
    if (S_out[2] && S_out[0]) { S_out[3]=0; S_out[1]=0; }
}

// Temperature Conversion Function for TMP36GZ
float ConvertTMP36ADCtoTemperature(uint16_t adc_raw_sensor) {
    float sensor_voltage = (adc_raw_sensor / ADC_MAX_VALUE) * ADC_VREF;
    // Apply formula: Temp_C = (Voltage - 0.5) / 0.01
    float temperature_C = (sensor_voltage - TMP36_OFFSET_V) / TMP36_SCALE_FACTOR_V_PER_C;
    return temperature_C;
}

// SWV Printf function
int _write(int file, char *ptr, int len) {
    (void)file;
    for (int i = 0; i < len; i++) {
        ITM_SendChar((uint32_t)(*ptr++));
    }
    return len;
}

void SWV_Printf(const char *fmt, ...) {
    va_list args;
    va_start(args, fmt);
    vsnprintf(printf_buffer, sizeof(printf_buffer), fmt, args);
    va_end(args);

    for (int i = 0; printf_buffer[i] != '\0' && i < (sizeof(printf_buffer) -1) ; i++) { // Check buffer limit
        ITM_SendChar((uint32_t)printf_buffer[i]);
    }
}
/* USER CODE END 4 */

/**
  * @brief  This function is executed in case of error occurrence.
  * @retval None
  */
```

```
void Error_Handler(void)
{
  /* USER CODE BEGIN Error_Handler_Debug */
  __disable_irq();
  // Example: Toggle an LED rapidly to indicate an error
  // Assuming LD2 (often PA5, but you re-used PA5) or another available LED
  // For instance, if you have an error LED on PB7 (User Label: Error_LED)
  // HAL_GPIO_WritePin(Error_LED_GPIO_Port, Error_LED_Pin, GPIO_PIN_SET);
  while (1)
  {
      // HAL_GPIO_TogglePin(Error_LED_GPIO_Port, Error_LED_Pin);
      // HAL_Delay(100);
  }
  /* USER CODE END Error_Handler_Debug */
}

#ifdef  USE_FULL_ASSERT
/**
  * @brief  Reports the name of the source file and the source line number
  *         where the assert_param error has occurred.
  * @param  file: pointer to the source file name
  * @param  line: assert_param error line source number
  * @retval None
  */
void assert_failed(uint8_t *file, uint32_t line)
{
  /* USER CODE BEGIN 6 */
  // SWV_Printf("Wrong parameters value: file %s on line %lu\r\n", (char*)file, line);
  /* USER CODE END 6 */
}
#endif /* USE_FULL_ASSERT */
```

Listing 8: The code inside of the STM board.