EMBEDDED SYSTEMS PROJECT REPORT

Processor-in-the-loop Implementation of Master-Slave Battery Management Systems

E-PiCo 2022-2024

Guillermo Israel Buenfil Solis

Gustavo Gomez Casanova

Musa Matthew

Bramantio Yuwono

Syed Umar Zia

8 April 2024

1. System Overview

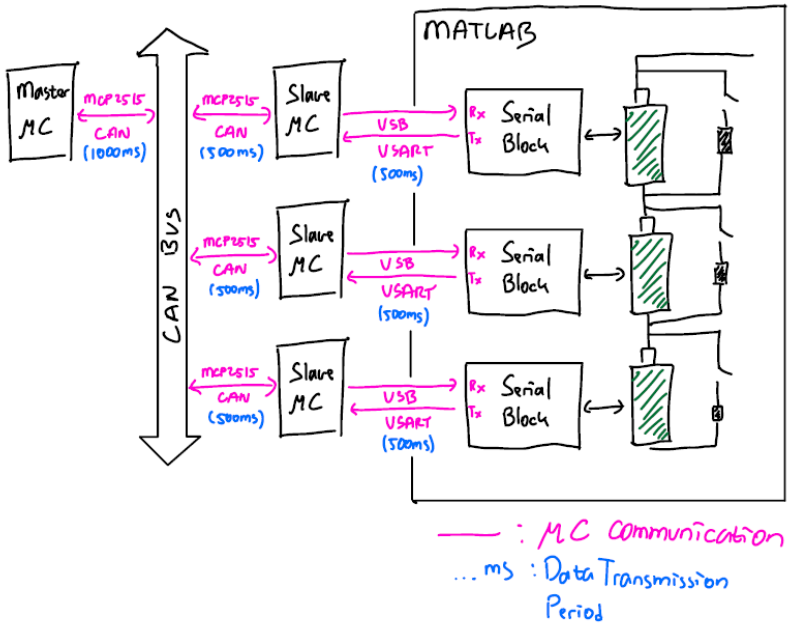

Figure 1 Overall System

The system's main purpose is to implement the voltage-based balancing controller without needing actual batteries connected to the battery management system. With this proposed method, the battery balancing algorithm can be verified without the risk of harming the actual batteries.

The battery cells and balancing circuit models are emulated on Simulink. Then, the models are interfaced with the battery management system utilising serial blocks. Serial blocks enable the data exchange between the controller implemented on Arduino Nano and the models on Simulink.

The Master-Slave BMS topologies are chosen for this project. In this topology, BMS slaves only read the battery parameters from the sensors (or in this case, the serial block) and pass the data to the BMS Master via CAN bus. Subsequentially, the BMS Master receives the battery parameters from all BMS Slaves and processes the data using the balancing algorithm to decide which cell requires to be balanced. Moreover, the BMS Master transmits the balancing command to each BMS slave. Based on the command from the BMS Master, BMS Slaves apply the passive balancing action (let the battery drain the charge through the resistor).

For this project, the transmission period for communication between the model and BMS Slaves, BMS Slaves to CAN bus, and the BMS Master to BMS Slaves via CAN bus are set to 500ms, 500ms, and 1s respectively. The effect of the transmission period on the balancing performance is exempted for this project.

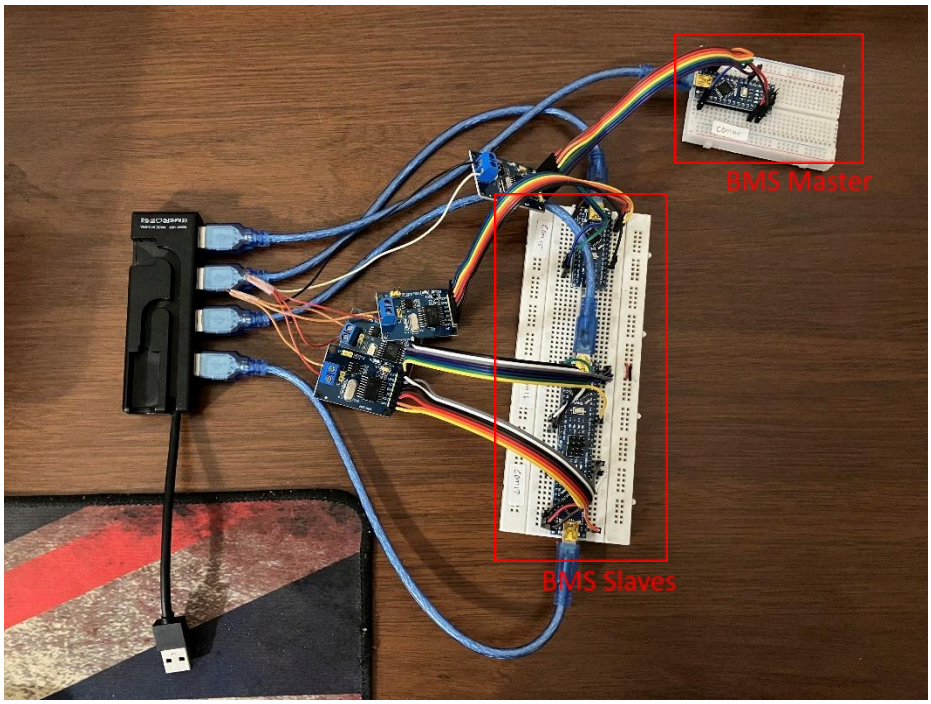

Figure 2 Real Implementation of BMS Master-Slave PIL

The implementation of the Master-Slaves BMS is shown in the figure above. All BMS modules are implemented on Arduino Nano. To enable CAN communication, MCP2515 is connected to each of the BMS modules. The connection between BMS Slaves and the models on Simulink is realized with USB to the USB hub. The USB connection between the BMS Master and the USB hub is made only for debugging purposes, not related to the system's main functionalities.

2. Implementation Details

This section gives a detailed explanation of the BMS modules algorithm, Simulink-Arduino interface, and battery cells & balancing circuit models. All related files can be referred to here.

a. Battery and Passive Balancing Models on Simulink

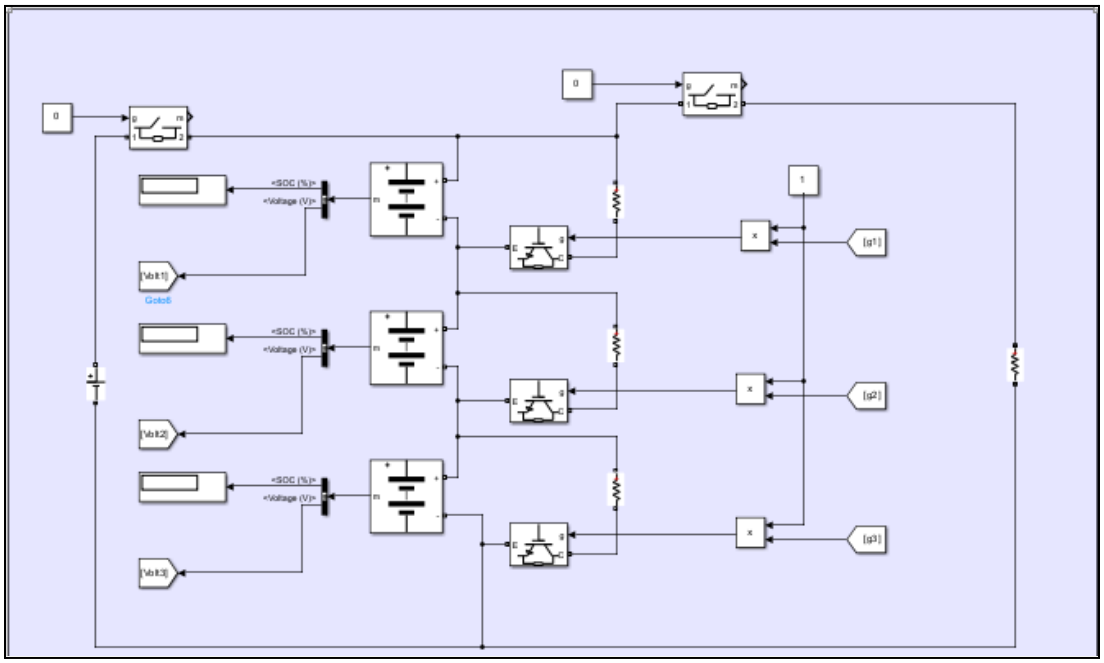

Figure 3 Balancing Battery Model

Initial SoC of batteries :
20, 25, 28 (can be changed in the battery blocks)

To balance : see function in the bottom left part.
g1, g2, g3 set to '1' when battery 1, 2, or 3 needs balancing (while charging).
So, we should implement that function in the arduinos rather than in Simulink, and receive the g's values from the comm part.

Figure 4 Balancing Battery Model Description

Balancing is performed to prevent the imbalance of the internal resistance, nominal voltage and temperature of the batteries. Also, when having batteries with different initial states of charge, one battery can reach 100 % SoC first than others, and if it keeps charging until the other batteries are fully charged, this first charged battery can suffer from an increase in operating temperature, endangering the battery lifespan. This same can

happen when one battery reaches 0% first and keeps discharging while others are still reaching zero furthermore balancing prevents this situation as well.

In Figure 3, we can see a balancing circuit model developed in Matlab which performs a balancing strategy for a balancing circuit which consists of a balancing switch (IGBT´s) and a balancing resistor together with a battery. This configuration is done for three batteries with different initial states of charge. The switches are then controlled by the suitable control system from the Arduino, which uses g1,g2, and g3 as the controlled variables.

The battery's chemistry is lithium-ion and has a nominal voltage of 3.2 volts. The battery capacity for the three batteries is 180 Ah and the initial SoC of the batteries are 20, 25, and 28 respectively in this image but can and will be changed to other SoC initial values later in this document for demonstration purposes.

As you can see in Figure 3, the balancing battery model uses a passive strategy. Each battery has a bleed or bypass resistor that is used to dissipate the excess energy (voltage) as heat and this resistor is connected through an IGBT switch. While charging; g1, g2, and g3 can be set to 1 to balance. This means that when the corresponding battery is reaching its full capacity the IGBT switch is turned on so the charge of this battery is removed and dissipated through the resistor, permitting all batteries to charge at the same time. The g1, g2, and g3 variables are the communication values sent and received to the SIL serial blocks of the communication model which will be better explained in the next part.

b. PIL Communication: Matlab Serial Block and Arduino Program

In order to make the connection between the Arduino and the balancing model, both parts had to be adequate to properly communicate one to the other.

In the Simulink program, a communication part dedicated to sending and receiving data from the Arduino was implemented. The main actors in this part are the user-implemented 'SIL Serial' blocks, as they collect the data that needs to be sent to the Arduino, as well as output the variables that the Arduino sends, both operations requiring proper data conditioning. They also allow us to configure the characteristics of the serial port. The operation of the 'SIL Serial' blocks relies on two existing blocks in Simulink: 'Serial Send', and 'Serial Receive'. As their names suggest, these blocks are in charge of the configuration and opening of an interface to the specified serial ports.

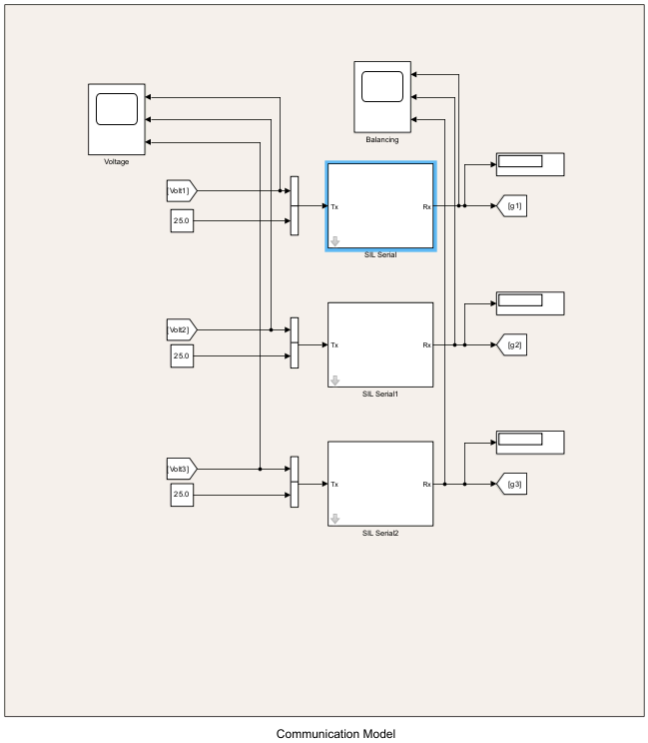

Figure 5 Communication model implemented in Simulink

A summary of the overall operation of the Simulink model is the following:

- The battery balancing circuit, which works at the same time as the serial communication, gives the value of the voltage of each battery to the 'SIL Serial' blocks. The data is conditioned and sent to the serial port previously specified. The control variables for the balancing circuit (g1, g2 and g3) are received in the same iteration and given back to the circuit, and the operation is repeated.
- In the Arduino program, two functions are created that allow us to communicate with the Simulink model: 'readFromMatlab()', and 'writeToMatlab()'. Both can be consulted in Figure 5. Their operation consists exclusively of reading/writing data to the serial port that the Simulink model is using to communicate.

```
203  bool readFromMatlab( FLOATUNION_t* f1, FLOATUNION_t* f2 ){
204    bool state = false;
205    FLOATUNION_t f;
206
207    // read the incoming bytes:
208    int rlen = Serial.readBytesUntil('\n', buff, BUFFER_SIZE);
209    for( int i=0;i<4;i++){
210      f1->bytes[i]=buff[i];
211    }
212    for( int i=4;i<8;i++){
213      f2->bytes[i-4]=buff[i];
214    }
215    state = rlen;
216
217    return state;
218  }
```

```
221  void writeToMatlab( FLOATUNION_t fnumber) {
222    // Print header: Important to avoid sync errors!
223    Serial.write('A');
224
225    for (int i=0; i<4; i++){
226            Serial.write(fnumber.bytes[i]);
227    }
228
229    Serial.print('\n');
230  }
```

Figure 6 User-defined function to communicate serially with the Simulink model.

c. Battery Management Master Algorithm Explanation

The BMS master is one of the major components of this project; it is a decision-maker who processes all the data coming from batteries and decides whether one of the batteries needs to be balanced or not. Once decided, it will give a command to a respective BMS slave to take the prescribed action.

The BMS master serves as the primary controller for the whole system. It relies on a communication protocol (CAN bus) to receive data from the BMS slave and send commands back. After receiving data values from the BMS slave, the BMS master performs specific computations before sending a command to the BMS slave to activate or deactivate cell balancing (updating the balancing state On/Off). This computation is essential for managing the health and state of charge of the battery.

MCP2515 CAN controller was used to handle the communication between the BMS master and the BMS slaves using a different identification number.

The following Arduino pins were used for CAN communication with the MCP2515 CAN controller chip:

- **VCC**: Connect to 5V on the Arduino Nano.
- **GND**: Connect to GND on the Arduino Nano.
- **CS (Chip Select)**: Connect to D10 (Digital Pin 10) on the Arduino Nano. This pin is used to select the MCP2515 module when communicating over SPI.
- **SO (Serial Out/MISO)**: Connect to D12 (Digital Pin 12) on the Arduino Nano. This pin is for the MCP2515 to send data to the Arduino Nano.
- **SI (Serial In/MOSI)**: Connect to D11 (Digital Pin 11) on the Arduino Nano. This pin is for the Arduino Nano to send data to the MCP2515.
- **SCK (Serial Clock)**: Connect to D13 (Digital Pin 13) on the Arduino Nano. This pin provides the clock signal for SPI communication.
- **INT (Interrupt)**: Connect to D2 (Digital Pin 2) on the Arduino Nano. This pin is used for handling interrupt-driven events from the MCP2515.

The BMS Master has several parameters which were also defined in the Arduino sketch as follows:

- **CAN_ID:** This is a unique identifier for the Slave module on the CAN bus which is usually assigned to a specific slave.
- **MasterID:** CAN ID of the BMS Master module.
- **CAN_Timeout:** Timeout value in milliseconds for transmitting CAN messages.
- **Cell_Bal:** Variable to store the current balancing state (On/Off).
- **Cell_Volt:** Variable to store the cell voltage reading.

These parameters were used to carry out the communication and cell balancing algorithm.

The BMS Master transmits a CAN message containing the following data to Slaves:

- **Balancing State (Byte 0)**: Indicates whether balancing is ON (0x01) or OFF (0x00) with a unique ID of BMS slave.

The BMS Master algorithm follows simple steps to perform the cell balancing task. The steps are explained as follows.

- **Receive CAN message**: The Master waits to receive CAN messages from the Slave.
- **Check the balancing state**: The Master computes the balancing states.
- **Prepare CAN message**: The Master constructs a CAN message with the following data: ***Slave ID*** and ***Balancing state***.
- **Transmit CAN message:** The Master transmits the constructed CAN message to the Slave module over the CAN bus.
- **Repeat the Process:** The Slave continuously repeats this process.

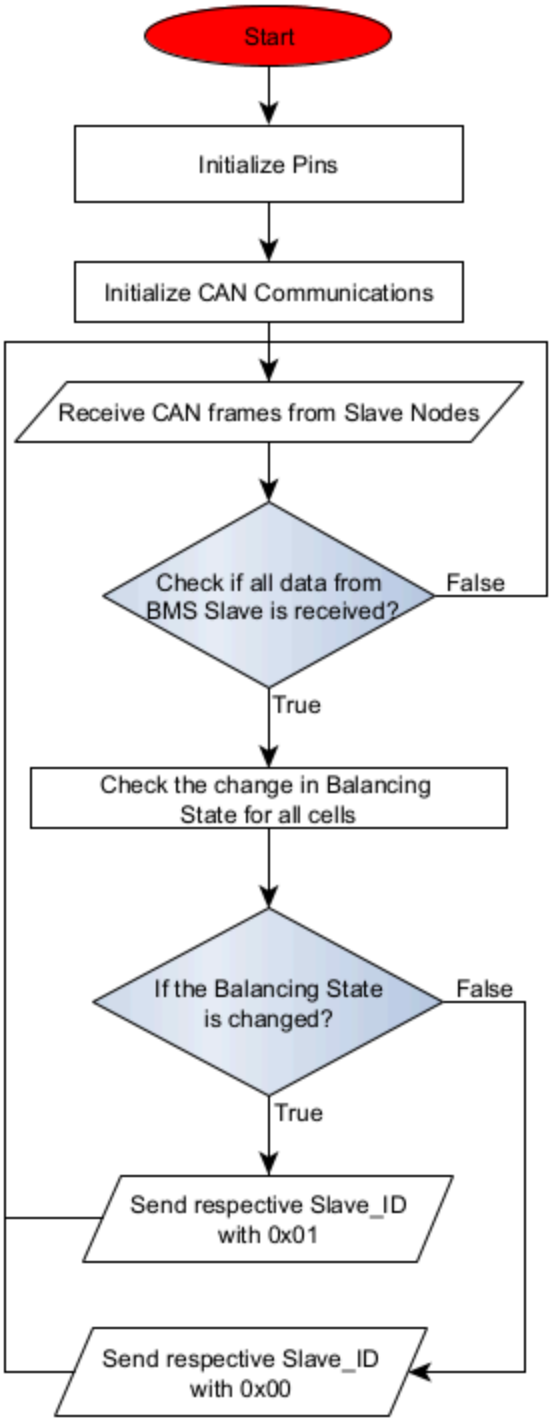

Figure 7 BMS Master flowchart.

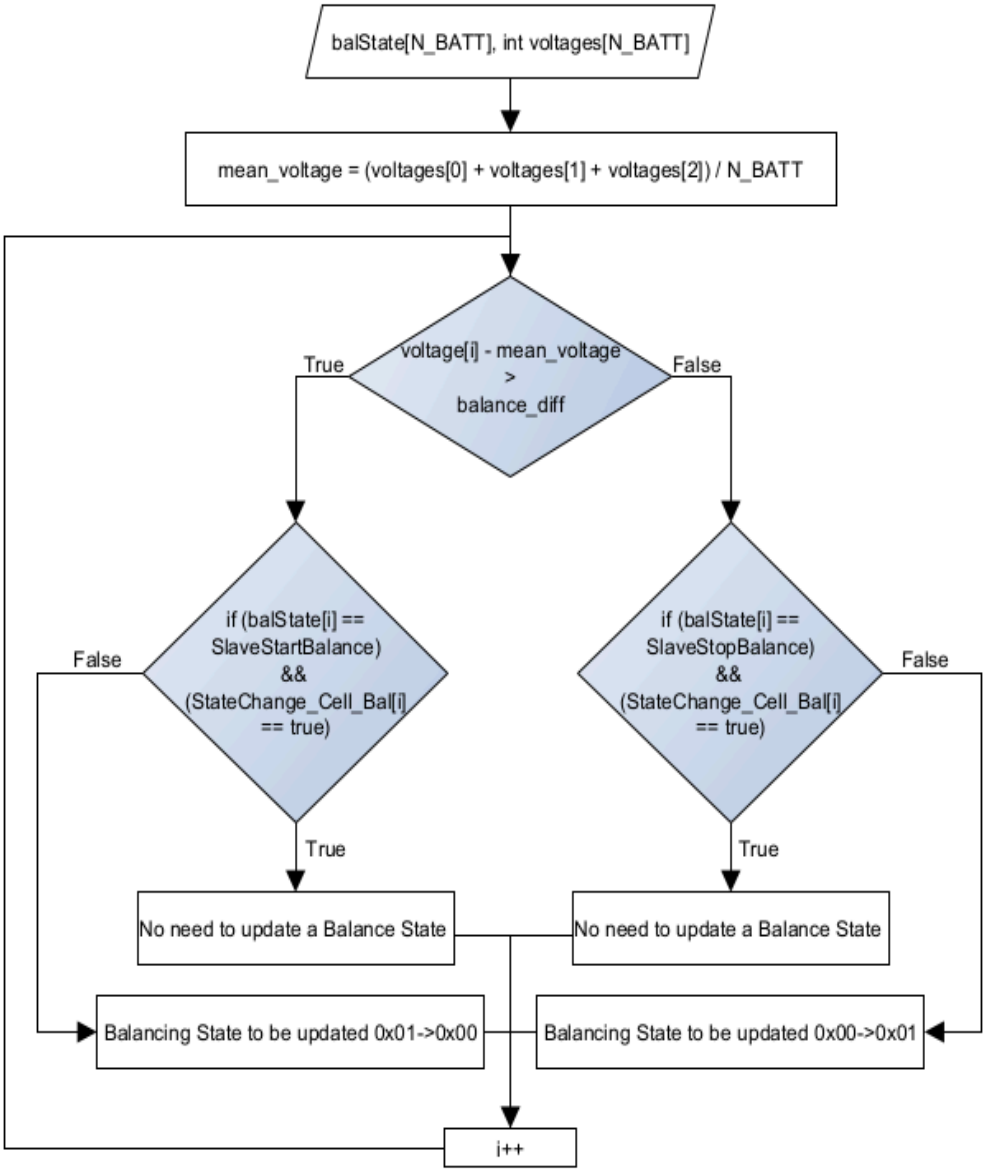

Figure 8 Update balancing state function flowchart.

The initialization process begins by setting up the necessary PINs for measuring cell voltage, CAN interrupt, and performing balancing operations. It is followed by initializing the CAN communication.

Subsequently, a CAN message is checked if it is received from the Slave BMS. When a message is received, it checks the incoming data against their maximum and minimum threshold. If it is within threshold or not the master BMS will send a command to the slave BMS to either

activate or deactivate the cell balancing process depending on the scenario. The process repeats.

d. Battery Management Slave Algorithm Explanation

The BMS slaves are one of the major components of this project; they handle the communication between the battery and the BMS Master, perform data acquisition, and carry out the balancing task.

BMS slaves serve as communication bridges between the BMS Master and individual battery cells. They utilize a communication protocol (CAN bus) to receive commands from the Master and transmit data back. In this project, the slave BMS Reads voltage values from each cell in the pack of the Simulink model through serial communication and sends it to the Master BMS. This information is essential for monitoring the health and state of charge of the battery. After receiving data from the BMS slaves, the BMS master will carry out particular computations and then send a command to the BSM slave to either activate or deactivate cell balancing (updating its balancing state On/Off). Balancing ensures all cells in the pack are at similar voltage levels, maximizing battery life and performance.

MCP2515 CAN controller was used to handle the communication between the BMS slaves and the BMS master using a different identification number. The connection between the BMS salve MCP2515 CAN controller involves interfacing the Arduino's SPI (Serial Peripheral Interface) pins with the CAN controller module. This setup allows the slave to communicate with the master via CAN bus systems.

The following Arduino (BSM Slave) pins were used for CAN communication with the MCP2515 CAN controller chip:

- **VCC**: Connect to 5V on the Arduino Nano.
- **GND**: Connect to GND on the Arduino Nano.
- **CS (Chip Select)**: Connect to D10 (Digital Pin 10) on the Arduino Nano. This pin is used to select the MCP2515 module when communicating over SPI.

- **SO (Serial Out/MISO)**: Connect to D12 (Digital Pin 12) on the Arduino Nano. This pin is for the MCP2515 to send data to the Arduino Nano.
- **SI (Serial In/MOSI)**: Connect to D11 (Digital Pin 11) on the Arduino Nano. This pin is for the Arduino Nano to send data to the MCP2515.
- **SCK (Serial Clock)**: Connect to D13 (Digital Pin 13) on the Arduino Nano. This pin provides the clock signal for SPI communication.
- **INT (Interrupt)**: Connect to D2 (Digital Pin 2) on the Arduino Nano. This pin is used for handling interrupt-driven events from the MCP2515.

Additionally, we also utilize the CANH and CANL PINs: These pins are on the MCP2515 module and were used to connect to the CANH and CANL wires of the CAN network. They are the differential signalling wires used for CAN communication between the BMS salves and the BMS master.

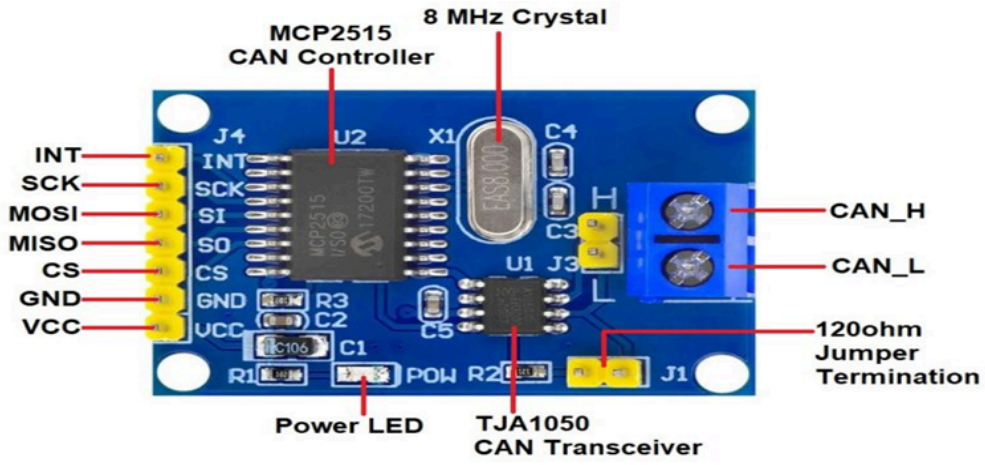

Figure 9 MCP2515 CAN Module

The BMS Slave has several parameters which were also defined in the Arduino sketch as follows:

- **CAN_ID:** This is a unique identifier for the Slave module on the CAN bus which is usually assigned to a specific slave
- **MasterID:** CAN ID of the BMS Master module
- **CAN_Timeout:** Timeout value in milliseconds for transmitting CAN messages
- **Cell_Bal:** Variable to store the current balancing state (On/Off)

- **Cell_Volt:** Variable to store the cell voltage reading

These parameters were used to carry out the communication and cell balancing algorithm.

The BMS Slave transmits a CAN message containing the following data to the Master:

- **Balancing State (Byte 0)**: Indicates whether balancing is ON (0x01) or OFF (0x00).
- **Cell Voltage (Bytes 1 & 2)**: a 16-bit value representing the cell voltage in millivolts.

The BMS Slave algorithm follows simple steps to perform the cell balancing task. The steps are explained as follows.

- **Receive CAN message**: The Slave waits to receive CAN messages from the Master
- **Update balancing state and perform cell balancing:** If a message is received from the Master, the Slave updates its internal balancing state variable (Cell_Bal) based on the received value (activate or deactivate cell balancing task)
- **Read voltage:** Reads voltage values from the Simulink model through serial communication.
- **Prepare CAN message**: The Slave constructs a CAN message with the following data: ***Balancing state*** and ***Cell Voltage***
- **Transmit CAN message:** The Slave transmits the constructed CAN message to the Master module over the CAN bus
- **Repeat the Process:** The Slave continuously repeats this process

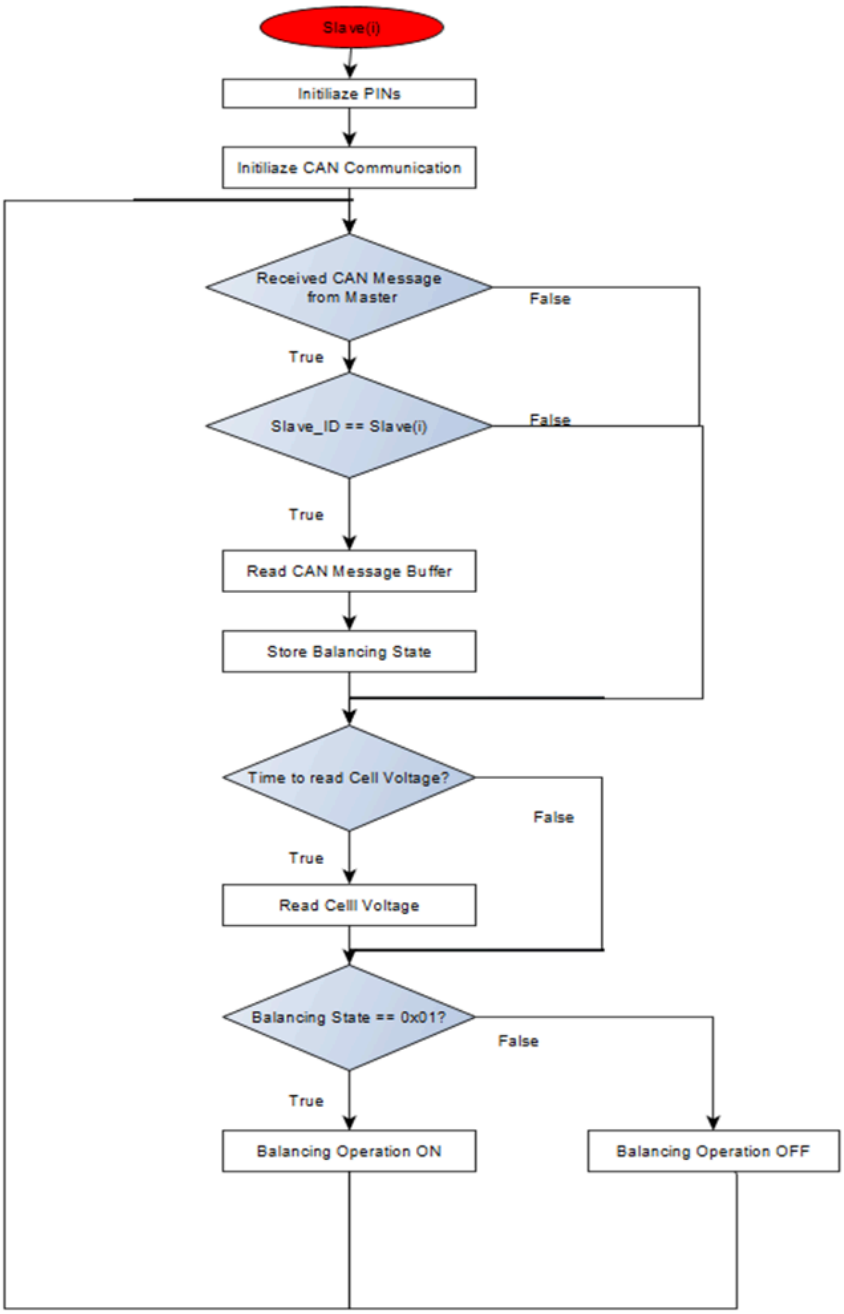

Figure 10 BMS Slave flowchart.

The initialization process begins by setting up the necessary PINs for measuring cell voltage, CAN interrupt, and performing balancing operations. It is followed by initializing the CAN communication.

Subsequently, a CAN message is checked if it is received from the Master BMS. If a message is received, it verifies if the intended message corresponds to a particular Slave using the ID number. If it matches, the transmitted cell balancing state will be stored. The obtained voltage cell and balancing state are stored in the CAN TxBuffer and transmitted to the Master BMS via CAN. Finally, the slave BMS will receive a command to either activate or deactivate the cell balancing process. The process repeats.

3. **Result and Observation**

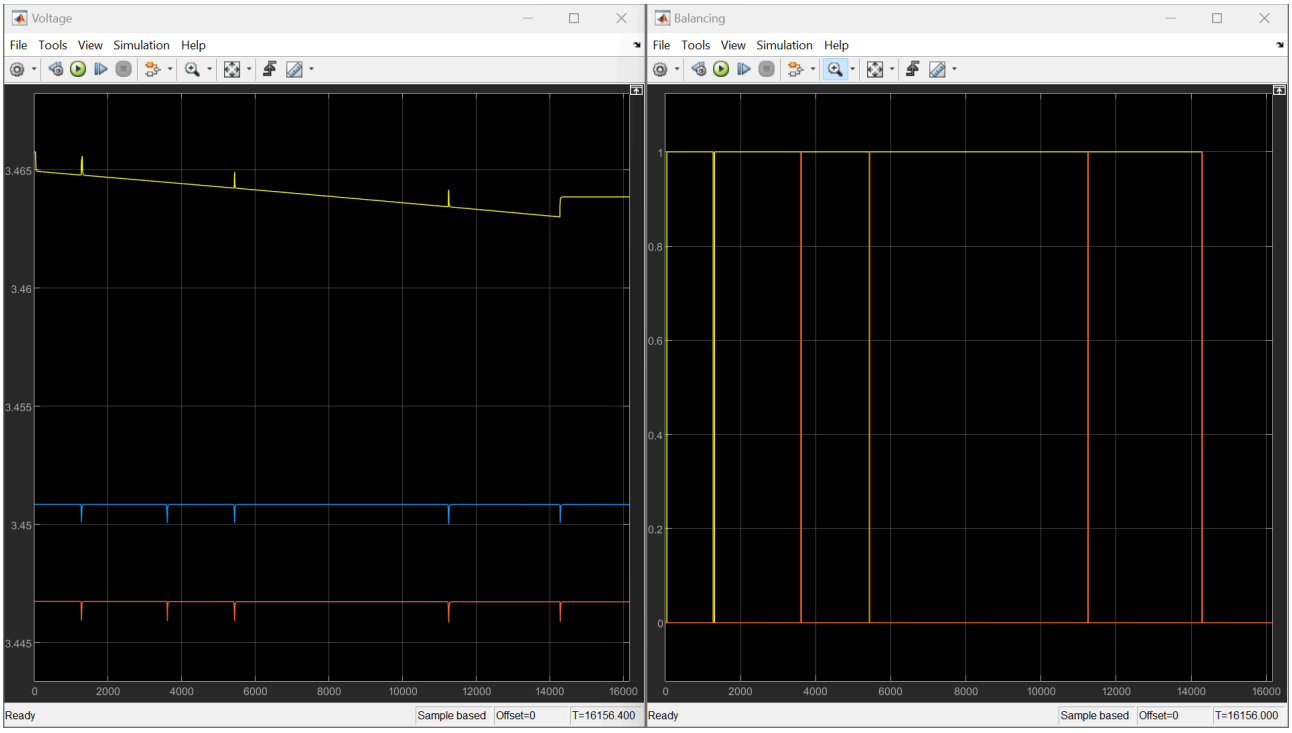

Figure 11. Battery Voltage Level(left) and Balancing State(right) during Testing

The test is performed while the battery is in an idle state (no charging or discharging) to ensure the observation of the balancing algorithm performance is not affected by the battery charging/discharging process. Initially, the SOC for each cell differed to 85% for cell 1 (yellow), 55% for cell 2 (blue), and 50% for cell 3 (red).

Generally, it can be observed that the balancing algorithm works well anticipating the voltage imbalance between three cells until the third cell voltage is reduced to the voltage level imbalance allowable range (5mv between each cell and the mean voltage of all three cell voltages).

However, in some cases, there are sudden drops in cell voltage likely due to the sudden change in the balancing command for cell 3. There are three possible causes of this: the misprocessing by the balancing algorithm, serial/CAN communication delay, and battery model abnormalities. Regarding this, further investigation is required.

4. Conclusion

From this project, it can be concluded into these points:

- The implemented processor-in-the-loop battery management system with master-slave topology for passive balancing can be emulated utilizing battery cell and balancing circuit models on Simulink connected to BMS modules via serial communication.
- Generally, the balancing algorithm on the BMS Master can identify which cells require to be balanced and send the command to the BMS Slaves. It can stop the balancing process also when the imbalance conditions are not fulfilled anymore.
- During testing, it is observed that there are sudden voltage drops in some cases. Further investigation on this is required to solve this issue.
- Further improvement for future works: SoC-based balancing algorithm, incorporating temperature into the battery model, investigation about the effect of CAN/serial transmission period on balancing performance, cell operating condition monitoring function.