Spring 2022

Zips Racing Electric Battery Management System

John R. Martaus
jrm245@uakron.edu

Elena Falcione
The University of Akron, emf66@uakron.edu

Derek Dunn
The University of Akron, dad157@uakron.edu

Elliott Boudreau
The University of Akron, eab126@uakron.edu

Custom Battery Management System

Senior Project Final Report

DT03

Elliott Boudreau

Derek Dunn

Elena Falcione

John Martaus

Md Ehsanul Haque

April 25, 2022

Table of Contents

List of Figures

List of Tables

Abstract:

Zips Racing Electric currently uses a bulky, off-the-shelf battery management system to monitor and manage the voltage, temperature, and state-of-charge of an electric formula-style racecar battery pack (accumulator). The main objective of this project is to research current battery management methodologies then apply said research to design and create a custom, lightweight, compact battery management system that is integrated with existing circuits. This will allow for more control over the accumulator, reduce wire harness bulk, and reduce weight of the car. Some of the main features are:

- Cell voltage monitoring
- Cell temperature monitoring
- Cell balancing
- State of charge tracking
- CAN bus communication

Author: DD, EF

Problem Statement

Need:

Zips Racing Electric builds an all-electric Formula-style racecar which is powered by a 420 V lithium-ion battery. Maintaining proper cell voltages and temperatures is critical to maintaining the health of the batteries and the safety of team members and bystanders. Currently, the off-the-shelf battery management system which monitors these parameters is bulky, with no compact way of packaging required overcurrent protection devices. This presents challenges to the design of the entire battery enclosure.

Authors: EF, DD

Objective:

The objective of this project is to design a custom battery management system (BMS). The solution must comply with the rules defined by SAE in EV.6 and EV.8. The BMS must monitor cell voltage and temperature, triggering a fault when these parameters go outside an acceptable range and shut down the racecar. The BMS must also be able to detect a fault in its own systems. The BMS will consist of a control board and sense boards. The control board will include a microcontroller, which must interface with the currently existing racecar system and complete the tasks described above.

Authors: JM, EB

Background:

The main function of a battery management system (BMS) is to keep batteries from operating outside of the manufacturer's specifications. The operation of batteries, especially the lithium-ion batteries used in the current design of the racecar, outside of the set parameters can lead to permanent damage or an explosion, which can be seen in Figure 1. For this reason, the battery cells must be monitored and managed continuously as to prevent any unwanted affects. There are four ways a typical battery management system does this: Temperature monitoring, current measurement, cell voltage monitoring, and cell balancing (Tarle et al., “Design of a Battery Management System”).

In order to maintain safety, the temperature of battery cells must be monitored continuously and must be kept below the operating temperature specified by the manufacturer. Failure to comply with this specification could result in a fire or an explosion (Duban et al., “Thermal Behavior”). The temperature also must be below 60°C as stated in SAE rule EV.8.5.2. The battery management system must shut down the racecar in the event of temperatures above the rated amount.

The current measurement system is used to ensure that the load for the system, the motor of the car in this case, is drawing an appropriate amount of current (Tarle et al., “Design of a Battery Management System”). If the current draw is large enough, which could be caused by a short between the positive and negative terminals of the battery, the temperature will spike dramatically causing thermal runaway (Duban et al., “Thermal Behavior”).

Author: JM

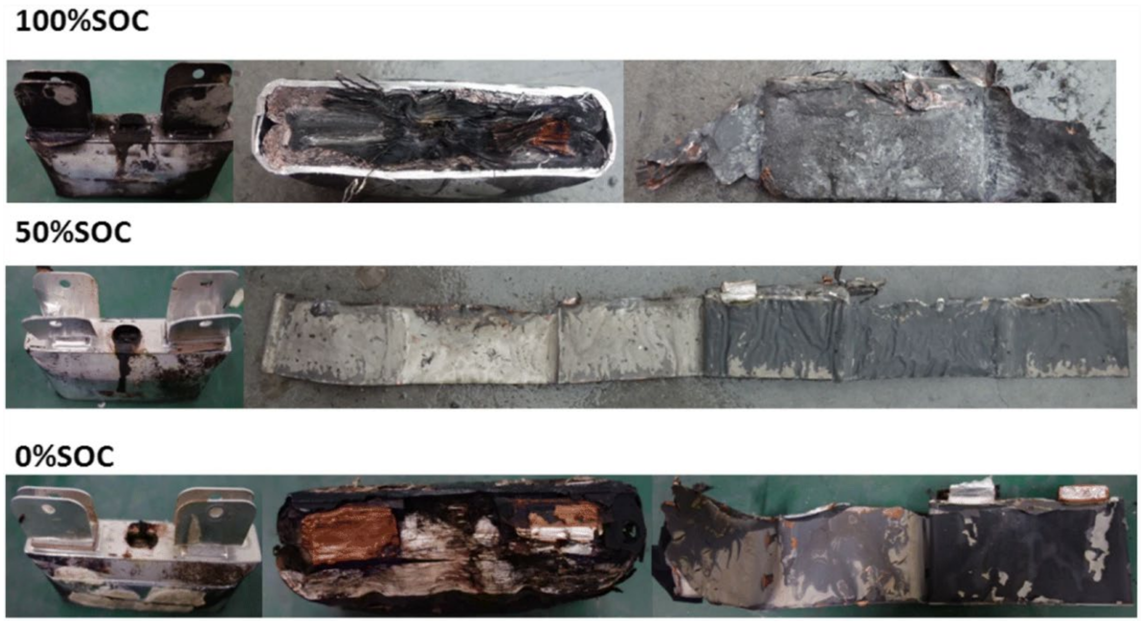


Figure 1: Result of Thermal Runaway for Various States of Charge as shown in "Thermal Behavior and Failure Mechanism of Large Format Lithium-Ion Battery."

Currently, Zips Racing Electric uses an off-the-shelf battery management system from Orion. There are two main limitations to this system which will be addressed in the new custom system.

The first limitation of the Orion BMS is its fixed isolation locations. The Orion BMS has three connectors for voltage sense lines. Each connector has 36 pins, which are divided into three groups. There is 100 V isolation between cell groups in the same connector, and 2.5 kV isolation between the connectors. This setup is illustrated by Figure 2 below. EV.6.1.1 specifies that the battery pack for Zips Racing Electric must be divided into segments which comply with certain energy and voltage limits. To comply with these limits, the divisions between battery pack segments must align with the 2.5 kV isolation boundaries in the BMS. This restricts Zips Racing Electric to a maximum of three segments in the battery pack. A custom BMS could be designed with isolation in separate locations to allow for any desired battery pack configuration.

Author: EF

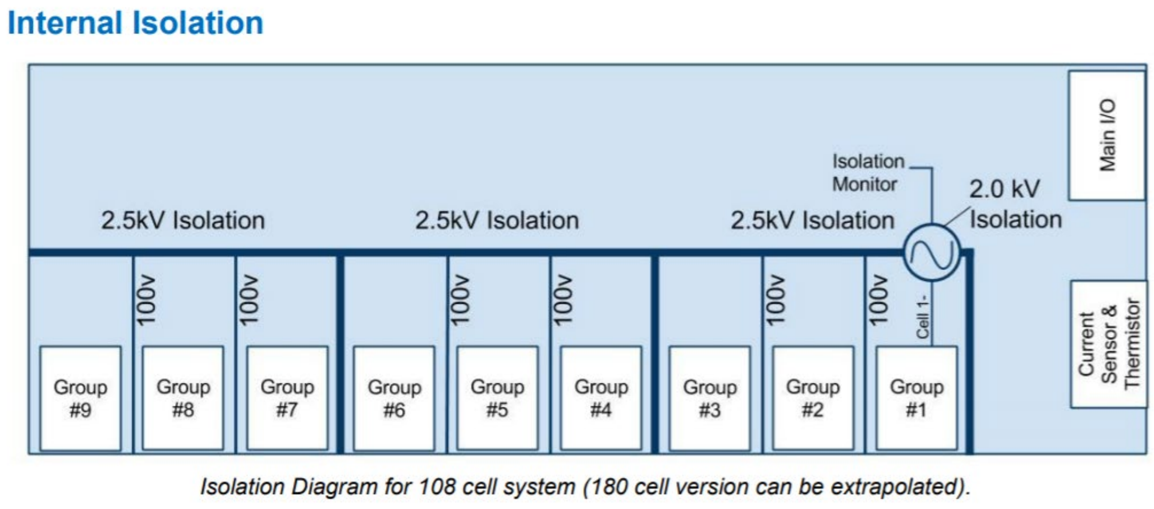


Figure 2: Orion BMS Isolation Diagram

The second limitation of the Orion BMS is the difficulty of packaging the required overcurrent protection devices. EV.8.4.3 requires that every voltage sense line include an overcurrent protection device and EV.8.3.3 requires that the BMS be capable of detecting when an overcurrent protection device has tripped. The vehicle must be safely shut down in this case. In previous years, Zips Racing Electric used in-line fuses mounted in spring-loaded cases to provide overcurrent protection. The spring-loaded cases are shown in Figure 3 below. The cases are black, and the wires connected to them are red. The racecar vibrates and drives over bumps, which can cause the spring-loaded cases to move inside the battery pack; this can occasionally result in one of the fuses momentarily losing contact with either side of the case. The Orion BMS will interpret this loss of contact as if the fuse had blown, and the vehicle will be shut down in compliance with EV.8.3.3. A custom BMS could be designed to use some other form of overcurrent protection, such as a surface mount fuse on a PCB, which would be more compact to package and less likely to result in a false BMS fault.

Author: EF

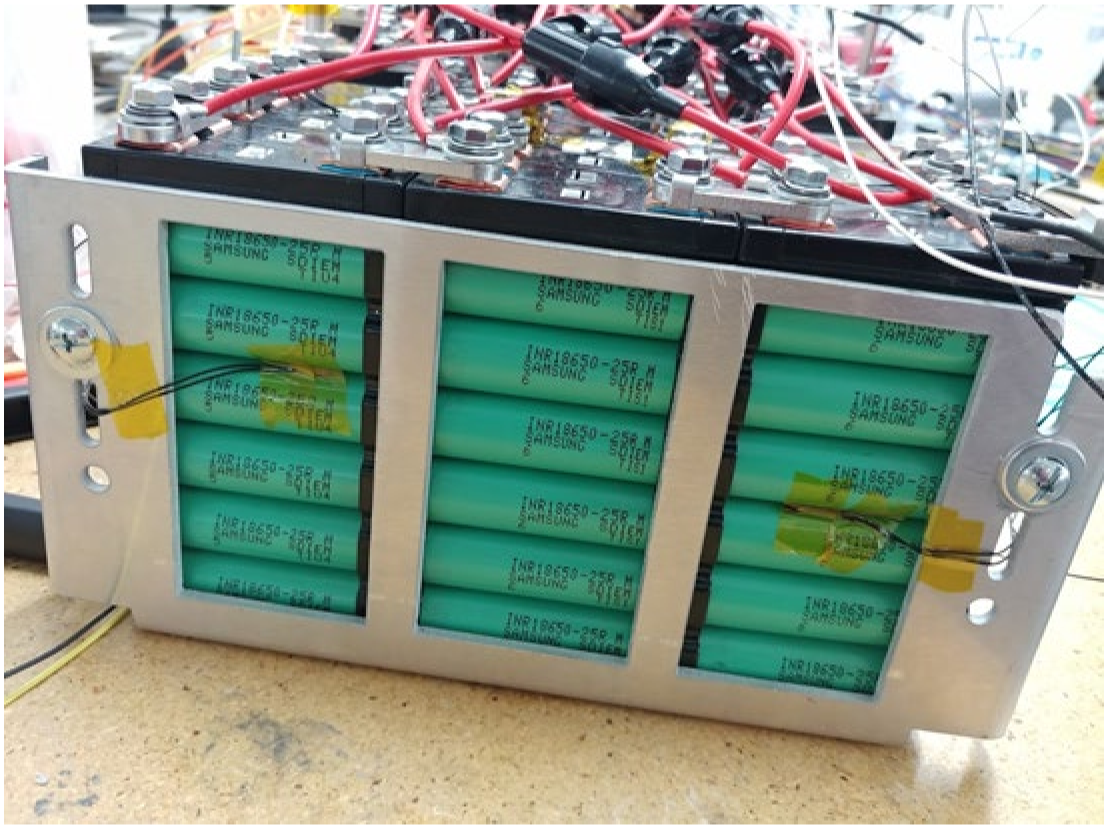


Figure 3: Black fuse holder with red wires

State of Charge calculation is an important function of a battery management system. State of Charge, or SoC, is a percentage which indicates how much charge is stored in a battery in relation to the total charge capacity of the battery. The charge stored in a battery and the total charge capacity of the battery are typically measured in Amp-hours. SoC can be used to determine remaining battery runtime before a recharge is necessary. This is important for Zips Racing Electric so the racecar driver can adjust their driving style, if necessary, to conserve energy during the 22 km endurance race. SoC is also an indication of a battery's overall health. Over many charge and discharge cycles, a battery's charge capacity will decrease. Even fully charged, a used battery may not be able to reach 100% SoC. This SoC degradation can be used to determine whether a battery requires replacement.

Author: EF

There are many ways to determine a battery's SoC, two of which will be discussed here. The simplest method is to use a lookup table which correlates the battery's voltage and its SoC. This method is easy to implement but not highly accurate. A plot of cell voltage vs. SoC for a lithium-ion battery is shown below. Typically, lithium-ion cell voltage remains nearly constant from 80% to 20% SoC, as shown in Figure 4. This makes it difficult to accurately determine battery SoC based on voltage alone (Pop, *State-of-the-Art of battery State-of-Charge determination* 11-16).

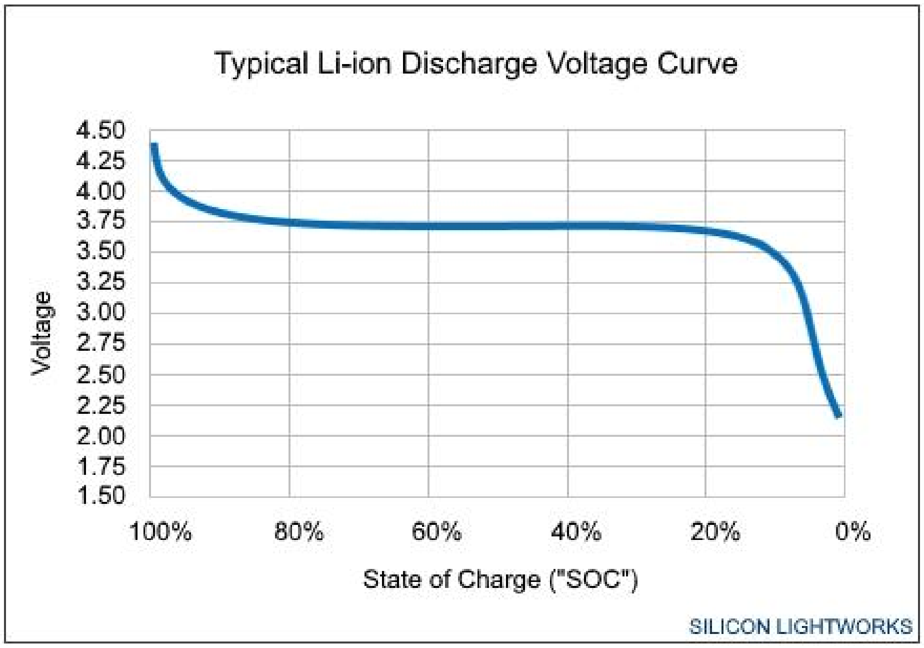


Figure 4: Typical Li-ion discharge curve

Another method to determine SoC is called Coulomb counting. Coulomb counting involves measuring current entering or exiting the battery and integrating with respect to time. In this way, the amount of charge entering or exiting the battery can be determined. This method requires calibration, while the simple lookup table method does not. A Coulomb counting algorithm must be calibrated by collecting data from a full charge (0 – 100% SoC) or discharge (100 – 0% SoC) cycle. This calibration process is used to measure how much charge must enter or exit the battery to travel from fully discharged to fully charged or vice versa. This allows the programmer to determine how much charge equates to 100% SoC. With this information, an equation for percent SoC as a function of charge can be determined, allowing the BMS to calculate SoC given any amount of charge (Pop, *State-of-the-Art of battery State-of-Charge determination* 11-16).

Author: EF

The battery management system will perform cell balancing to prevent undercharging and overcharging of cells, as well as ensure that the voltage of each cell is uniform. As described by Andrei Vladimirovich Ivanov's patent, *Accumulator battery management system,* there are two widely used categories of cell balancing: active and passive. A system that utilizes active cell balancing will keep track of the voltage of each cell during a charge or discharge cycle. If a cell's neighboring cells have a lower voltage, the cell with higher voltage will discharge a portion of its potential to its neighbors, increasing their potential. This process will repeat until the voltage of the cell and its neighboring cells are the same. The passive cell balancing system will also keep track of the voltage of each cell, but instead of passing the excess potential of one cell to other cells, the excess is discharged over a load resistor, which produces heat. The active method is more efficient than the passive method, but it is more complicated to design (Ivanov, *Accumulator battery management system*).

Author: DD

The cell balancing method will determine the capacity of the battery that results from combining all the cells. A battery that is managed using the active cell balancing strategy will have a maximum capacity equal to the average capacity of the cells it is made up of, while a battery that is managed using the passive cell balancing strategy will have a maximum capacity equal to the maximum capacity of the lowest capacity cell in the battery (Ivanov, *Accumulator battery management system*).

Author: DD

Ivan Loncarevic's patent, *Battery management system with temperature sensing and charge management for individual series-connected battery cells*, is a relevant example of a battery management system that uses a passive cell balancing system. In his design, each cell of the battery has its own control unit with a passive balancing circuit that shunts the cell with a load resistor if the control unit determines that the cell is being over charged. All the cell control units are controlled by a central control unit which keeps track of the state of each cell (Loncarevic, *Battery management system with temperature sensing and charge management for individual series-connected battery cells*).

Author: DD

Active cell balancing techniques employ methods to preserve the unbalanced potential, instead of simply dissipating the energy into heat. The first example of active cell balancing to be considered is the capacitive balancing method. This method utilizes either one or a series of capacitors as a temporary charge storage system. In a single Capacitor system, a micro controller is used to transfer the unbalanced potential from any given cell with a high SOC to the capacitor by connecting the two in parallel. The capacitor charges and is then connected in parallel with a cell with a low SOC. This method is known as direct cell to cell. As a method it requires few components, while being able to function in high power applications. The system falls short due to its low speed and high controller complexity. This low speed and high system complexity can be improved with the addition of more capacitors. (Hemavathi, *Overview of Cell Balancing Methods FOR Li-Ion Battery Technology*).

Author: EB

Another more efficient method of cell balancing utilizes transformers to balance the cells. The two under consideration are the shared transformer method and the switched transformer method. The switched transformer utilizes current from the entire battery pack to power cells with low SOC by transferring current through a transformer into an intelligent set of switches. The switches will disperse the potential to the cells with lower SOC. This method is similar to the single capacitor system; however, it balances twice as fast. The capacitive system used the switching controller to transfer power to a capacitor only for the same system to route the potential later. The switching transformer method only requires the potential to get sent to the low SOC cells, while the power comes from the entire cell pack at once. This simplifies the control system required to operate the switches as well as the losses through the switches. If a designer wishes to use zero control systems to balance the cells actively, the shared transformer method will be an acceptable option. The shared transformer refers to a single transformer with the primary tied to the high and low end of the pack, while secondary taps are split amongst each of the cells. Thus, the cell with the lowest terminal voltage will have the most induced current, balancing the cells. This method requires no intelligent switching; however, it is quite complicated and thus costly to build a transformer to achieve this circuit. (Welsh, *A Comparison of Active and Passive Cell Balancing Techniques for Series/Parallel Battery Packs*).

Author: EB

Whether or not the active balancing method utilizes a capacitor or an inductor as the energy storage device, there are five methods of moving potential around the battery pack. These methods are Cell Bypass, Cell-to-Cell, Cell-to-Pack, Pack-to-Cell, and Cell-to-Pack-to-Cell. Cell Bypass systems balance the cells by bypassing cells that reach maximum or minimum voltages until the other cells also reach the same levels. These systems are useful for charging and discharging cycles only and are not useful for balancing cells in any other operation. Cell-to-Cell method refers to systems like the single capacitor active balancing system. Where cells that have a high SOC transfer charge to cells with a low SOC. Cell-to-Pack systems and Pack-to-Cell systems are similar in nature, you transfer charge from high SOC cells to the entire pack, or take charge from the whole pack and disperse it to cells with low SOC. The two-transformer based balancing methods are both pack to cell. Cell-to-Pack-to-Cell methods are simply a combination of the two Cell-to-Pack and Pack-to-Cell methods, these systems tend to yield the most efficiency, but require the most components and are thus the priciest. The BMS project will most likely run on an active system involving capacitors, due to their size and cost, while implementing a Cell-to-Cell method. This way the system is not overly complicated while achieving acceptable efficiency and speed. (Omariba, *Review of Battery Cell Balancing Methodologies for Optimizing Battery Pack Performance in Electric Vehicles*).

Author: EB

Marketing Requirements:

1. BMS must be packaged in the existing battery pack more easily than the current Orion off-the-shelf solution.
2. BMS must interface with the existing vehicle electrical system.
3. BMS must monitor battery voltage and temperature.

4. BMS must perform cell balancing.

5. BMS must comply with the safety rules described by SAE in EV.6 and EV.8.

Authors: EF, EB

Engineering Analysis

Circuits:

In this section, the engineering analysis for the control board (AMS) and sense boards circuits will be discussed.

Control Board – DC/DC Regulators

The maximum current draw for all the components operating at +3.3V and+5V is listed in the tables below.

+3.3V Supply			
Component	Function	I_{MAX} (mA)	Total (mA)
DSPIC33EP128GS804-I/PT	Microcontroller	300	330.5
(2) MCP2562-E/SN	CAN Transceiver	0.5	
AT25040B-XHL-T	EEPROM	10	
LTST-C190KRKT	LEDs	20	

Table 1: Max current for +3.3V rail

+5V Supply			
Component	Function	I_{MAX} (mA)	Total (mA)
DHAB S/118	Current Sensor	20	227
(2) MCP2562-E/SN	CAN Transceiver	140	
LMC7221BIM5X/NOPB	Comparator	40	
LTC6820IMS#PBF	BMS Interface	7	
LTST-C190KRKT	LEDs	20	

Table 2: Max current for +5V rail

The DC/DC regulators (R-78E3.3-1.0 and R-78E5.0-1.0) chosen are rated to a maximum of 1A. The efficiency graphs of the regulators are shown below in Figure 5.

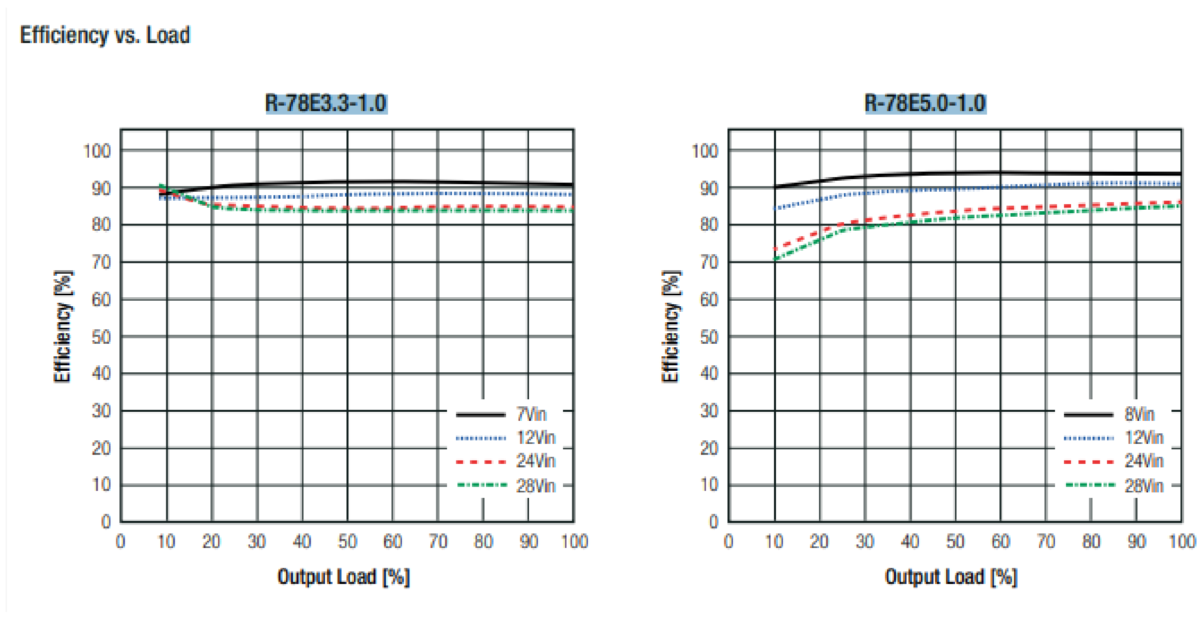


Figure 5: Efficiency vs. Load for the DC/DC Regulators

For the +3.3V regulator, the output is load is roughly 33% which correlates to an efficiency of about 87%. Half of a watt will be lost as heat due to the efficiency loss at the rated current.

$$\frac{100\% - 87\%}{100} * (12V * 330.5mA) = 0.52W\ Lost$$

For the +5V regulator, the output load is roughly 23% of the max which correlates to an efficiency of about 87%. There is a max of 0.35W power lost as heat to the efficiency rating at the max current.

$$\frac{100\% - 87\%}{100} * (12V * 227mA) = 0.35W\ Lost$$

Control Board – Memory

The memory chip or electrically erasable programmable read-only memory (EEPROM) stored data even on power down (Non-volatile). However, as a power down is occurring, no data should be written to the EEPROM. To disable writing to the EEPROM, the write protect pin must be pulled low.

For normal operation:

$$Inverting\ Input: 5V * \frac{10k}{10k + 10k} = 2.5V$$

$$Non - Inverting\ Input: 5V * \frac{10k}{10k + 26k} = 3.33V$$

For normal operation, the inverting input is smaller than the non-inverting input, so the output is pulled high. This disables the write protect and allows the microcontroller to write to the memory.

For power down situations:

$$Non - Inverting\ Input: 9V * \frac{10k}{10k + 26k} = 2.5V$$

When the system is powering down, the non-inverting is lower than the inverting which pulls the output of the comparator low and enables write protect.

Electronics:

The Battery Management System will be modeled in Simulink. The Simulink toolboxes Simscape and State Flow will be used in conjunction to create an accurate model. This model will have the ability to monitor cell voltages, cell state of charge, cell current, and cell temperatures. The model will actively balance the cells and monitor fault conditions, triggering a shutdown state if needed.

Authors: EB

The battery management system will measure the current discharged through the accumulator and the temperature of 20% of the cells will have their temperatures measured. The accumulator current will be measured with a current transducer. The current transducer will output a DC voltage that relates to the current by measuring the magnetic field from the wire. The temperature will be measured with a series of thermistors. Thermistors are a type of resistor that is heavily dependent on temperature. Each thermistor is allowed to monitor multiple cells, for the thermistors will only be used to verify the cells remain below 60 degrees C. The thermistors will be used as the bottom of a voltage divider with a constant resistance in the upper half of the voltage divider. The voltage measured at the output of the voltage divider will be monitored by the BMS.

Authors EB

The battery management system model needs to control an accurate model of the battery. This battery will be known as the accumulator and will consist of a collection of cells in series and parallel. The cells in practice will have varying charge and discharge characteristics. The Voltage vs SoC curves are also nonlinear, adding extra complexity. In order to model the complexity a simple equivalent circuit is used. The circuit has an ideal voltage source, an in-series resistance, and one or two RC transient blocks. This model is chosen as a compromise between complexity and simulation ease. To find the values for the resistances and capacitances of the components, a cell discharge test must take place.

Authors: EB

The Cell discharge test requires some knowledge of the Voltage vs. SoC discharge curve of a Lithium-Ion battery. The regular Lithium-Ion battery cell has a non-linear discharge curve. The cell tends to vary voltage around the fully charged and fully discharged states. When in the 90% to 10% SoC region, the cell will maintain a steady voltage. This linear region makes testing the cells difficult. The cells must have their current measured and a coulomb counting method must be used. Thus, a test using a BK Precision 8500 series programable load bank will be conducted. The cells will be discharged for around one minute with a one-hour rest interval. This rest interval is necessary due to the nature of Li-Ion battery chemistry. This discharge/rest cycle will be run until the cell is fully discharged. This Voltage vs SoC curve is imported into Matlab as a lookup table. This lookup table is then compared with a simple example curve generated by the equivalent circuit component values. These values are then adjusted using the parameter estimation function of Matlab. This function marginally adjusts values until it matches the experimental curve.

Authors: EB

Once the Accumulator is modeled the BMS is created using State Flow. State Flow will create a state machine which will use the inputs from the cells (voltages, currents, SoC, and temperature) to balance and fault monitor the cells. The State Flow model will work in three different modes (Charging, Discharging, and Idle). These modes will output a series of curves showing cell voltages and temperatures. With these curves our BMS system will be perfectly testable without any physical components.

Authors: EB

Communications:

The BMS will communicate with other systems in the car by way of the CAN communication protocol, as all other systems in the vehicle currently use CAN to communicate. CAN will also be used internally in the BMS to transmit all data collected to a data logger. CAN is a digital communication protocol which transmits series of bits from one device to another. CAN is robust against electrical noise and often used in automotive applications. At the physical level, CAN consists of a pair of wires, often referred to as CAN High and CAN Low. These two wires normally float at around the same voltage. To transmit a bit 0, called a dominant state, CAN High is driven to a logic level high, and CAN Low is driven to a logic level low. To transmit a bit 1, called the recessive state, the lines are allowed to float back to their resting voltage. The Zips Racing Electric car CAN bus operates at 1 MB, or 1 million bits per second. While CAN bus does not include any error correction methods, it does include error detection in the form of a CRC. CAN messages consist of several overhead bytes and up to 8 bytes of data as shown in Figure 6.

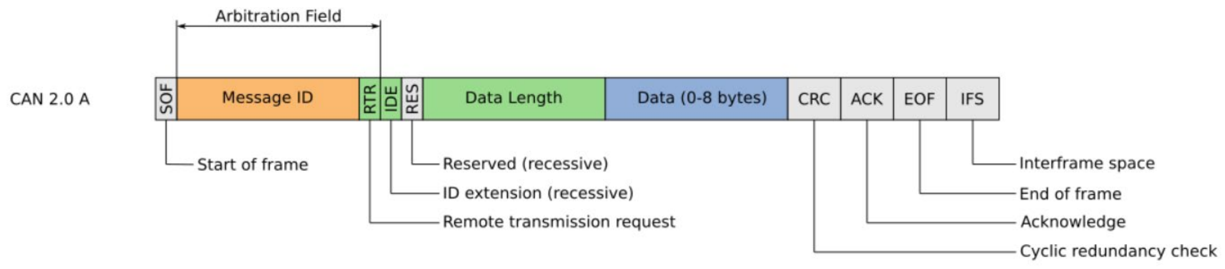


Figure 6: CAN Bus Message Structure

A CAN bus device will wait to begin transmission until the bus is idle. If two CAN bus devices begin transmission at the same time, message arbitration occurs. The device transmitting the message with the higher ID field will defer to the other device. The device which deferred will wait until the bus is idle again to attempt retransmission. In this way, message priority can be adjusted by selecting message IDs intelligently. Higher priority messages should be assigned lower IDs.

Author: EF

SPI communication typically assigns one device as the primary and the rest as secondary devices. Devices share a clock signal, a MOSI line, and a MISO line. The primary device outputs information onto the MOSI line. The secondary devices can respond by outputting information on the MISO line. When multiple secondary devices are used, a chip select line is connected from the primary to each of the secondary devices and is used to indicate which device is being addressed in a particular message on the MOSI line.

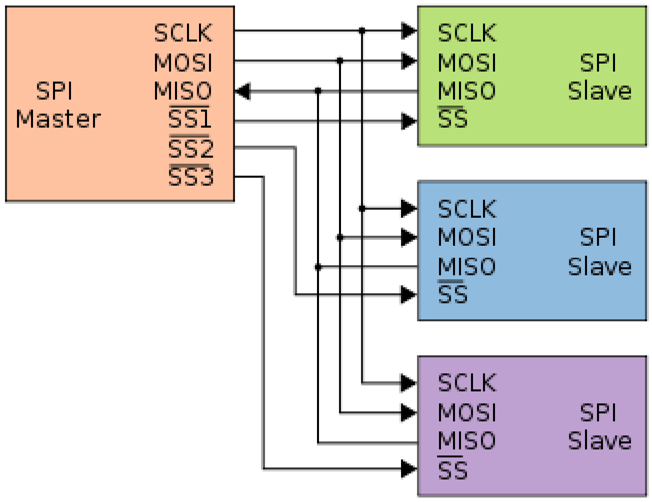


Figure 7: SPI Communication Connections

One of the engineering requirements is that 420 V isolation be maintained between the control board and the sense boards. To comply with this requirement, isolated SPI will be used for communication between the sensing boards and the BMS microcontroller. IsoSPI does not require a clock signal to be connected between devices. The sensing boards are connected in a daisy-chain configuration, to eliminate the need for chip select lines for each sense board. An isoSPI transceiver manipulates the MOSI and MISO signals into a differential signal pair. The resulting lines are named IP and IM. Transformers are used to achieve isolation of the IP and IM lines.

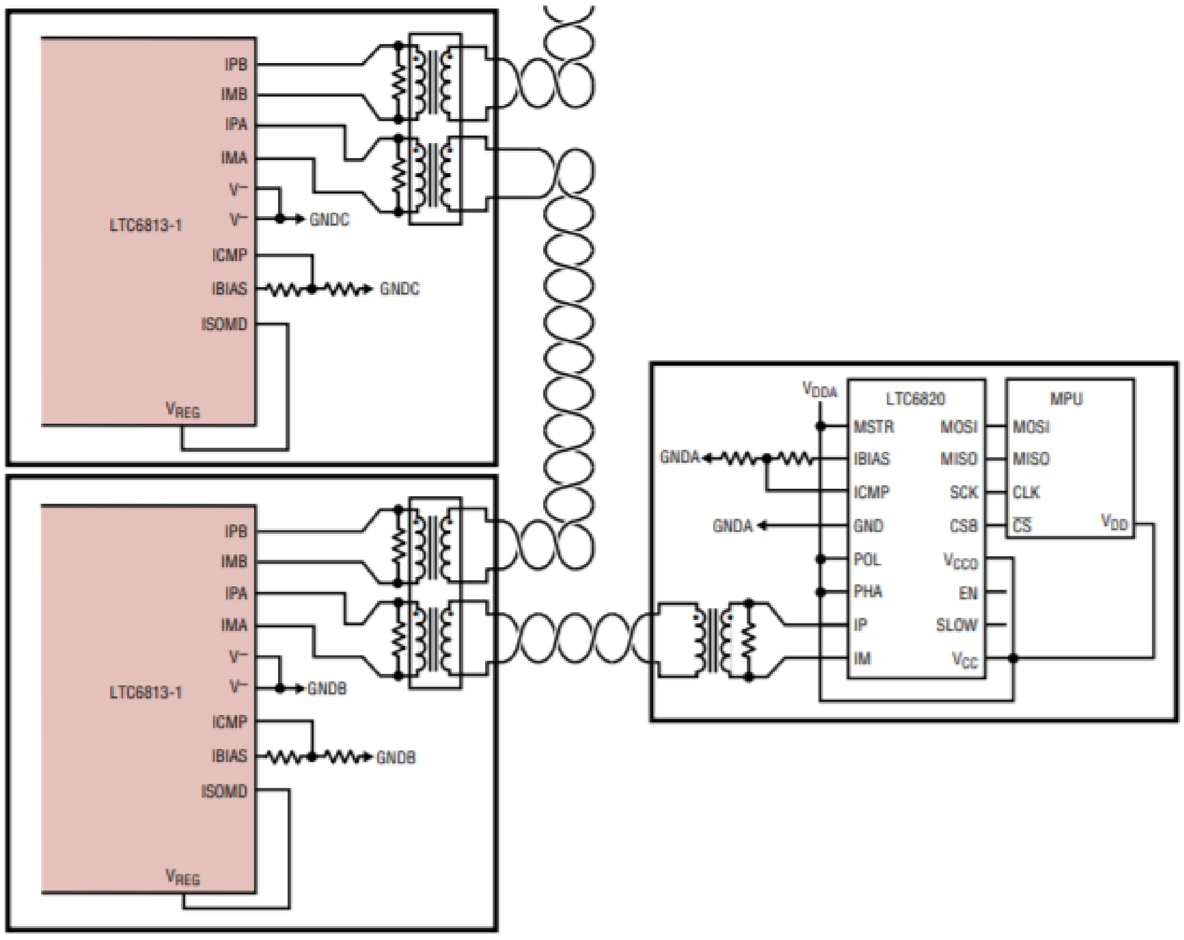


Figure 8: Daisy-chain isoSPI Configuration

Author: DD, EF

Embedded Systems:

Zips Racing Electric has determined they will construct an accumulator out of 90 battery cells series with 7 in parallel divided into five segments. The BMS must monitor at least 20% of the cell temperatures.

$$20\% * 630\ cells = 126\ cells$$

The BMS must monitor cell temperatures across all five segments.

$$\frac{126\ cells}{5\ segments} \approx 26\ cells/segment$$

The BMS must monitor the voltage of every cell in the pack. Therefore, the BMS chip used for monitoring a single segment must have a minimum of 18 pins for voltage sensing and enough pins to measure the temperature of at least 26 cells per segment. According to the rules, a single temperature sensor can be used to measure multiple cells.

Authors: JM

Zips Racing Electric uses PIC microcontrollers for other purposes on the racecar, so for consistency the BMS will also use a PIC microcontroller. It was determined that a minimum of 42 pins are required, by the following analysis:

- Two CAN bus: 6 (CAN TX/RX, Chip Standby)
- PC Interface: 2 (RS232 TX/RX)
- SPI: 8 (BMS chips, external memory)
- External Interfacing (From other PCBs): 10
- Debug LEDs: 4
- External oscillator: 2
- Programming pins: 2
- Power/ground: 8

Authors: EF

Engineering Requirements Specification

Engineering Requirement	Related Marketing Requirement	Rule
BMS must monitor the voltage of all battery cells and balance all cells with 5 mV resolution within 5% tolerance	3, 4	EV.8.4.1 - The AMS must measure the cell voltage of every cell
BMS must monitor the temperature of at least 20% of battery cells within 5% tolerance	3	EV.8.5.5a - The temperature of a minimum of 20% of the cells must be monitored by the AMS
BMS must include at least 420 V of isolation between battery segments	5	EV.8.3.2 - The AMS must have galvanic isolation at every segment-to-segment boundary.
BMS must include a self-test feature to detect internal faults	5	EV.8.3.4e – The AMS must monitor for a fault in the AMS
BMS must be able to detect if the overcurrent protection device on a voltage sense line has tripped within 500 ms	5	EV.8.4.3 - All voltage sense wires to the AMS must Have Overcurrent Protection
BMS must communicate fault conditions to the driver in real time	2,5	EV.8.3.6 - The AMS Indicator Light must be red, labeled “AMS” and clearly visible to the driver
BMS must shut down vehicle within 500ms when a fault occurs	2,5	EV.8.3.5 – If the AMS detects a fault, the car must shut down
BMS must track battery pack State of Charge in real time	N/A	N/A
BMS must provide timestamped fault information to the vehicle data logger via CAN bus	2	N/A
BMS must communicate battery SOC and temperature with vehicle ECU via CAN bus	2	N/A
BMS must be able to display measured voltage, temperature, current, and state of charge by connecting a laptop to the BMS	2	FSG EV.5.8.11 - The AMS must be able to read and display all measured values

Engineering Standards Specification

Safety:

Lithium-ion batteries can be dangerous to work with if safety precautions are not put in place. Any damage to or misuse of the cell can cause leaks, fires, and explosions. Many scenarios can cause the cell to rapidly disassemble including puncturing, shorting the terminals, over-charging, over-discharging, and operating outside the allowable temperature range.

If any of these scenarios are to occur, the cells should be smothered in sand and the fire department should be called immediately. When work on the cells is taking place, proper personal protective equipment (PPE) should be worn at all times. Proper PPE includes high voltage rated gloves and tools and safety glasses. A bucket of sand should also be kept near the work site. Most importantly, no team member should be working with the battery cells by themselves.

Author: JM

Communication:

Table 3 shows the format of the CAN messages to be sent by the BMS. In order to communicate all the cell voltages and temperature measurements, many CAN messages must be produced. To avoid bogging down the main vehicle CAN bus, these messages are not sent on the main vehicle CAN bus. They are transmitted on the BMS dedicated CAN bus. A single CAN message containing the battery state of charge, highest cell temperature, and fault information will be shared with the main vehicle CAN bus so that other vehicle devices can react if necessary. Relevant information will be communicated to the driver via the vehicle dashboard when this CAN message is received.

Table 4 shows the isoSPI message formats used for internal BMS communication. The read cell voltage commands (RDCVx) and the ADC conversion commands (ADxx) will be used to collect cell voltage information. The read auxiliary group commands (RDAUXx) and the ADC conversion commands (ADxx) will be used to obtain cell temperature information. The ADOW command will be used to satisfy the engineering requirement that the BMS detect if a voltage sense line becomes disconnected. The WRCFGA and WRCFGB commands will be used to command certain cells balance switches to conduct. The isoSPI commands are constructed using a command code, bits to indicate whether to read or write the affected registers, and a packet error code (PEC) which can be used for error detection.

Author: EF

Data Formats:

Data will be sent to the CAN data logger then stored in a text file. The CAN messages have the following structure:

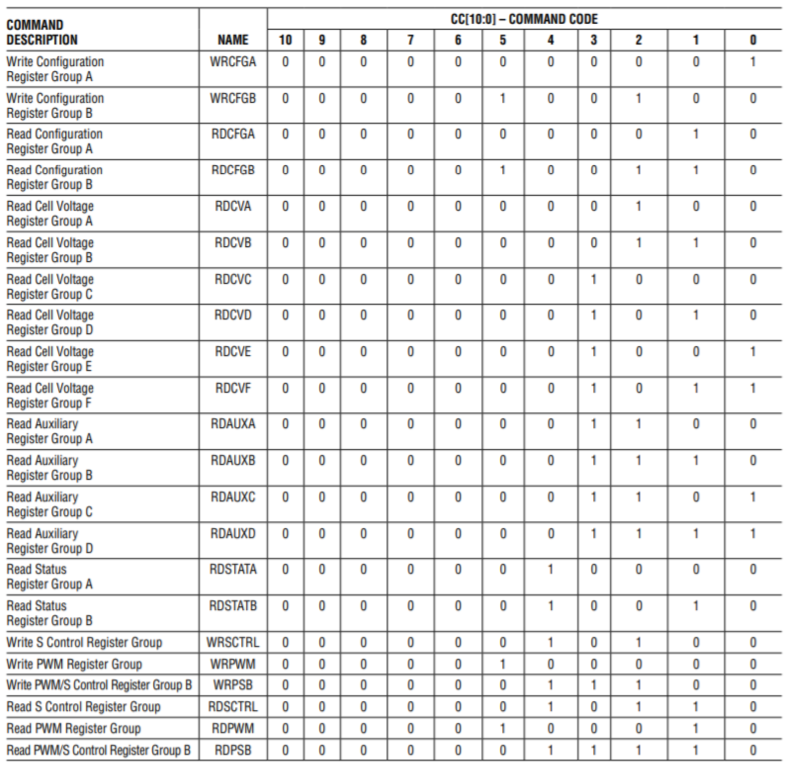
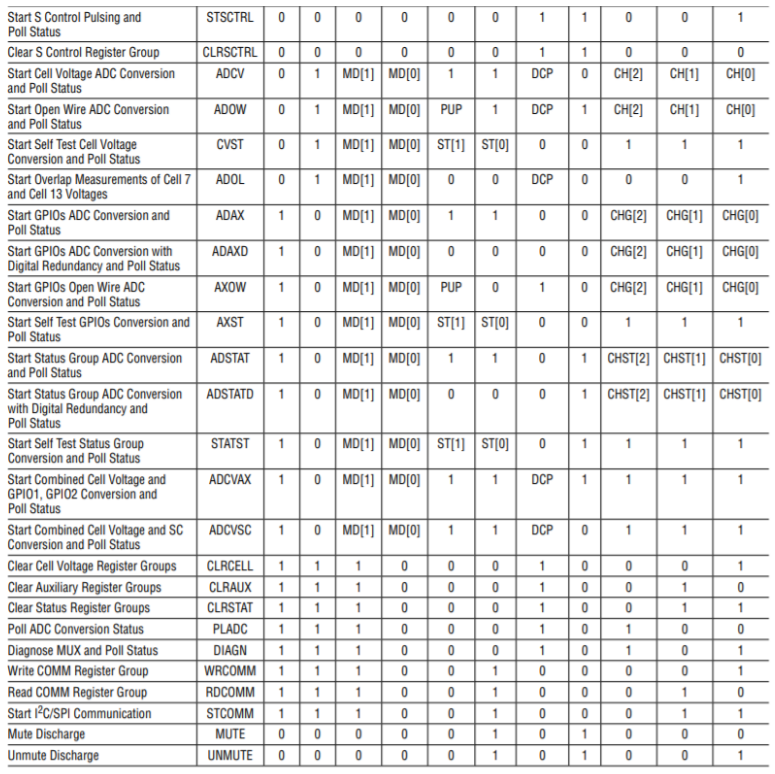
	ID	DLC	Data0	Data1	Data2	Data3	Data4	Data5	Data6	Data7	rate (ms)	notes
BMS CAN bus to GUI	0x401-0x410	8	cell0 voltage		cell1 voltage		cell2 voltage		cell3 voltage		20	repeat 22 times, V*10
	0x411	4	cell0 voltage		cell1 voltage		▨	▨	▨	▨	20	V*10
	0x412	8	fuse blown ID 0	fuse blown ID 1	fuse blown ID 2	fuse blown ID 3	fuse blown ID 4	fuse blown ID 5	fuse blown ID 6	fuse blown ID 7	500	number cell whose voltage sense line fuse is blown
	0x413	8	cell balancing bit field 0-63								100	flags are set to 1 if cell is balancing
	0x414	4	cell balancing bit field 64-89				▨	▨	▨	▨	100	
	0x415-0x41F	8	temp0		temp1		temp2		temp3		20	repeat 11 times, C*10
	0x420	2	temp0		▨	▨	▨	▨	▨	▨	20	C*10
	0x440	4	SoC	pack current		fault code	▨	▨	▨	▨	20	%, A*10, bit field

Main CAN bus	0x100	8	high-temp	power	SoC	pack voltage		fault code	pack current		100	C, kW, %, V*10, bit field, A*10

Table 3: CAN Message Formats

Author: EF

The BMS communicates internally via isoSPI. The isoSPI messages consist of a command code and bits to indicate whether a read or write operation is desired. The command codes have the following format:

COMMAND DESCRIPTION	NAME	CC[10:0] – COMMAND CODE										
		10	9	8	7	6	5	4	3	2	1	0
Write Configuration Register Group A	WRCFGA	0	0	0	0	0	0	0	0	0	0	1
Write Configuration Register Group B	WRCFGB	0	0	0	0	0	1	0	0	1	0	0
Read Configuration Register Group A	RDCFGA	0	0	0	0	0	0	0	0	0	1	0
Read Configuration Register Group B	RDCFGB	0	0	0	0	0	1	0	0	1	1	0
Read Cell Voltage Register Group A	RDCVA	0	0	0	0	0	0	0	0	1	0	0
Read Cell Voltage Register Group B	RDCVB	0	0	0	0	0	0	0	0	1	1	0
Read Cell Voltage Register Group C	RDCVC	0	0	0	0	0	0	0	1	0	0	0
Read Cell Voltage Register Group D	RDCVD	0	0	0	0	0	0	0	1	0	1	0
Read Cell Voltage Register Group E	RDCVE	0	0	0	0	0	0	0	1	0	0	1
Read Cell Voltage Register Group F	RDCVF	0	0	0	0	0	0	0	1	0	1	1
Read Auxiliary Register Group A	RDAUXA	0	0	0	0	0	0	0	1	1	0	0
Read Auxiliary Register Group B	RDAUXB	0	0	0	0	0	0	0	1	1	1	0
Read Auxiliary Register Group C	RDAUXC	0	0	0	0	0	0	0	1	1	0	1
Read Auxiliary Register Group D	RDAUXD	0	0	0	0	0	0	0	1	1	1	1
Read Status Register Group A	RDSTATA	0	0	0	0	0	0	1	0	0	0	0
Read Status Register Group B	RDSTATB	0	0	0	0	0	0	1	0	0	1	0
Write S Control Register Group	WRSCTRL	0	0	0	0	0	0	1	0	1	0	0
Write PWM Register Group	WRPWM	0	0	0	0	0	1	0	0	0	0	0
Write PWM/S Control Register Group B	WRPSB	0	0	0	0	0	0	1	1	1	0	0
Read S Control Register Group	RDSCTRL	0	0	0	0	0	0	1	0	1	1	0
Read PWM Register Group	RDPWM	0	0	0	0	0	1	0	0	0	1	0
Read PWM/S Control Register Group B	RDPSB	0	0	0	0	0	0	1	1	1	1	0
Start S Control Pulsing and Poll Status	STSCTRL	0	0	0	0	0	0	1	1	0	0	1
Clear S Control Register Group	CLRSCTRL	0	0	0	0	0	0	1	1	0	0	0
Start Cell Voltage ADC Conversion and Poll Status	ADCV	0	1	MD[1]	MD[0]	1	1	DCP	0	CH[2]	CH[1]	CH[0]
Start Open Wire ADC Conversion and Poll Status	ADOW	0	1	MD[1]	MD[0]	PUP	1	DCP	1	CH[2]	CH[1]	CH[0]
Start Self Test Cell Voltage Conversion and Poll Status	CVST	0	1	MD[1]	MD[0]	ST[1]	ST[0]	0	0	1	1	1
Start Overlap Measurements of Cell 7 and Cell 13 Voltages	ADOL	0	1	MD[1]	MD[0]	0	0	DCP	0	0	0	1
Start GPIOs ADC Conversion and Poll Status	ADAX	1	0	MD[1]	MD[0]	1	1	0	0	CHG[2]	CHG[1]	CHG[0]
Start GPIOs ADC Conversion with Digital Redundancy and Poll Status	ADAXD	1	0	MD[1]	MD[0]	0	0	0	0	CHG[2]	CHG[1]	CHG[0]
Start GPIOs Open Wire ADC Conversion and Poll Status	AXOW	1	0	MD[1]	MD[0]	PUP	0	1	0	CHG[2]	CHG[1]	CHG[0]
Start Self Test GPIOs Conversion and Poll Status	AXST	1	0	MD[1]	MD[0]	ST[1]	ST[0]	0	0	1	1	1
Start Status Group ADC Conversion and Poll Status	ADSTAT	1	0	MD[1]	MD[0]	1	1	0	1	CHST[2]	CHST[1]	CHST[0]
Start Status Group ADC Conversion with Digital Redundancy and Poll Status	ADSTATD	1	0	MD[1]	MD[0]	0	0	0	1	CHST[2]	CHST[1]	CHST[0]
Start Self Test Status Group Conversion and Poll Status	STATST	1	0	MD[1]	MD[0]	ST[1]	ST[0]	0	1	1	1	1
Start Combined Cell Voltage and GPIO1, GPIO2 Conversion and Poll Status	ADCVAX	1	0	MD[1]	MD[0]	1	1	DCP	1	1	1	1
Start Combined Cell Voltage and SC Conversion and Poll Status	ADCVSC	1	0	MD[1]	MD[0]	1	1	DCP	0	1	1	1
Clear Cell Voltage Register Groups	CLRCELL	1	1	1	0	0	0	1	0	0	0	1
Clear Auxiliary Register Groups	CLRAUX	1	1	1	0	0	0	1	0	0	1	0
Clear Status Register Groups	CLRSTAT	1	1	1	0	0	0	1	0	0	1	1
Poll ADC Conversion Status	PLADC	1	1	1	0	0	0	1	0	1	0	0
Diagnose MUX and Poll Status	DIAGN	1	1	1	0	0	0	1	0	1	0	1
Write COMM Register Group	WRCOMM	1	1	1	0	0	1	0	0	0	0	1
Read COMM Register Group	RDCOMM	1	1	1	0	0	1	0	0	0	1	0
Start I^2C/SPI Communication	STCOMM	1	1	1	0	0	1	0	0	0	1	1
Mute Discharge	MUTE	0	0	0	0	0	1	0	1	0	0	0
Unmute Discharge	UNMUTE	0	0	0	0	0	1	0	1	0	0	1

Table 4: isoSPI Message Format

Author: EF

When the BMS is connected to the PC monitoring application, the data will be stored in a SQL database. The database will include five tables: an accumulator data table, which will record data that pertains to the overall accumulator (state of charge, total voltage, current draw, highest cell temperature reading, and a flag indicating if a fault has been thrown); a cell data table, which will record data that pertains to each cell (the id of the cell, a timestamp, the voltage of the cell, a flag indicating if the cell is balancing, and a flag indicating if the cell fuse has blown); a faults table which will be used as a lookup table. This table will contain fault codes and descriptions associated with the fault; a faults thrown table, which will record any faults that are thrown and when they were thrown; and a temperature table, which will record the cell temperature readings collected by the thermistors.

Author: DD

Design Methods:

Agile SCRUM and Gantt methodologies will be used to ensure that design and productions schedules are on time. By using these methodologies, the product quality will be evaluated after every design review. Design reviews will occur every two weeks.

Author: JM

The model view viewmodel (MVVM) software architectural pattern will be used to design the BMS monitoring application. This pattern splits the codebase into three sections: the model, view, and viewmodel. The model is the data layer. This is where data will be modified and stored. The view is the user interface (UI) layer. This is where the application layout and UI controls are defined. The viewmodel is the bridge between the model and the view. In this layer, the data from the model is bound to the controls and views in the view. Following this design pattern allows the UI code to be written independently from the backend code, which allows changes to be made to the backend without affecting the frontend and vice-versa.

Author: DD

Programming Languages:

C will be used to program the embedded systems in the BMS. C# will be used in conjunction with WinUI 3 to create the monitoring application used to view live BMS data from a PC. MySQL will be used to store all data sent to the PC when using the monitoring application.

Author: DD

Connector Standards:

The VAL-U-LOK brand of connector from the manufacturer TE Connectivity will be used for all off-board connections within the accumulator. This brand of connector is standard to the rest of the car, so no crimping tools will need to be purchased. For connections from the accumulator to the car, watertight connectors from Molex will be utilized. The connectors must protect the wire harness from the environment.

Application	Standard	Use
Safety	Formula SAE 2022 Rule Book	The Formula SAE rules state how the car must be built and operate. All aspects of the car must follow the rules in order for it to be race eligible.
Communication	ISO 11898 ISO/IEC 14776-115	The ISO standards for CAN and SPI must be used to ensure correct communication between all devices on the car.
Data Formats	ISO/IEC 9075	ISO describes the framework needed for database management. The standards must be followed in order to ensure that data is stored correctly so that it can be read.
Design Methods	Agile SCRUM, Gantt	To ensure a quality product is delivered on time, agile SCRUM and Gantt will be used to keep design and production schedules on time.
Programming Languages	ANSI C	All C code should follow the ANSI C standard to ensure that it is readable and functional.
Connector Standards	Formula SAE 2022 Rule Book	Connectors must follow the SAE standards as required in Formula SAE rule book.

Author: JM

Accepted Technical Design

Hardware Design:

Level 0 – Hardware Diagram and Functional Requirements Table

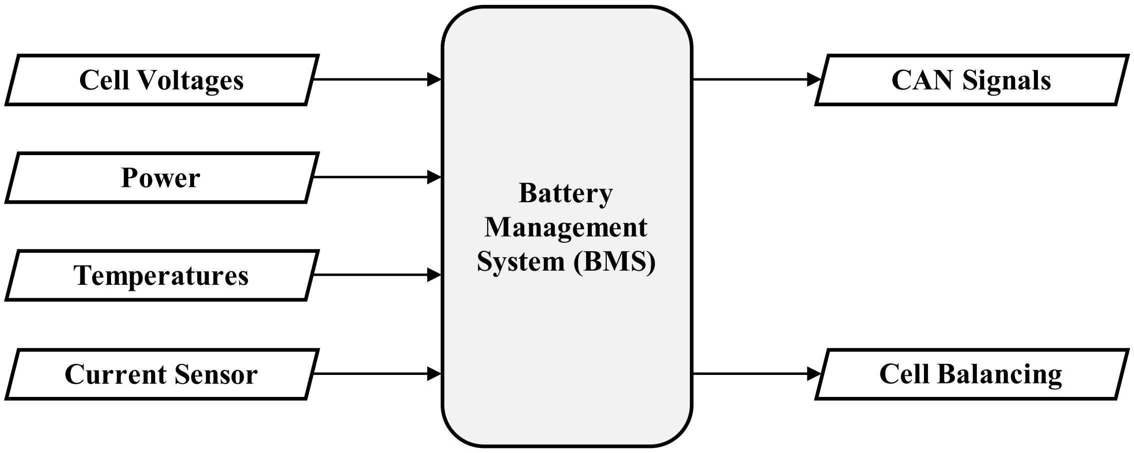


Figure 9: Level 0 Hardware Block Diagram

Module	Level 0: Battery Management System (BMS)
Designers	John Martaus, Elliott Boudreau, Elena Falcione, Derek Dunn
Inputs	Cell Voltages: Monitors voltages of all cells. Max 4.2V Power Supply: +12V Cell Temperatures: Monitors 20% of cells Current Sensor: Monitors current. 0-220A
Outputs	CAN Signals: 1MBaud Voltage Balancing: Maintain balanced voltages between cells
Description	Balances cell voltages and monitors cell voltages, temperatures, and tractive system current, and outputs the information via CAN bus.

Table 5: Level 0 Functional Requirements

The level 0 block diagram for the battery management system shows that the system must receive data from the current sensor, temperature sensors, and voltage sense lines. The circuit must be powered by 12V. Cell balancing must also be completed by the battery management system to ensure that the lithium-ion batteries operate at the same voltage. All data critical data will be sent to the vehicle CAN bus.

Author: JM

Level 1 – Hardware Diagram and Functional Requirements Tables

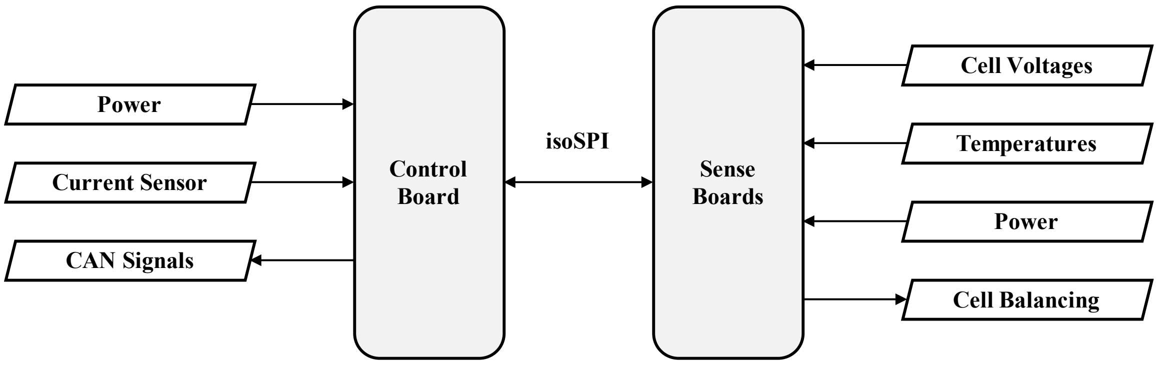


Figure 10: Level 1 Hardware Block Diagram

Module	Level 1: Control Board
Designers	John Martaus, Elliott Boudreau
Inputs	Sense Data: isoSPI sent from contol board. Power Supply: +12V Current Sensor: Measures tractive system current. Max 60˚C
Outputs	Control Data: SPI data sent to the sense board. CAN Signals: 1MBaud
Description	The control board sends control signals to the sense boards. The sense boards require instructions to be sent to them in order to function. The contoller also reads the tractive system current.

Table 6: Level 1 Control Board Functional Requirements

Module	Level 1: Sense Board
Designers	John Martaus, Elliott Boudreau
Inputs	Cell Voltages: Monitors voltages of all cells. Max 4.2V Cell Temperatures: Monitors 20% of cells Control Data: SPI sent from contol board
Outputs	Sense Data: isoSPI data sent to control board Voltage Balancing: Equalizing cell voltages.
Description	The sense board monitors the cell voltages of all the cells and the temperature of 20% of the cells. It must also balance all the cells. The sense data is sent to the control boards.

Table 7: Level 1 Sense Boards Functional Requirements

The battery management system is split into the control board and the sense boards. The microcontroller is located on the control boards. Commands and data will be sent and received via isoSPI between the control board and the sense boards. Power is supplied by the low voltage system battery, and the sense boards are powered by the cells that it monitors. CAN data is outputted from the control board. Current sensor data is sent directly to the microcontroller to be converted from an analog signal to a digital signal. Cell voltages and temperatures are monitored via the sense boards. The sense boards also control cell balancing.

Author: JM

Level 2 – Hardware Diagram and Functional Requirements Tables

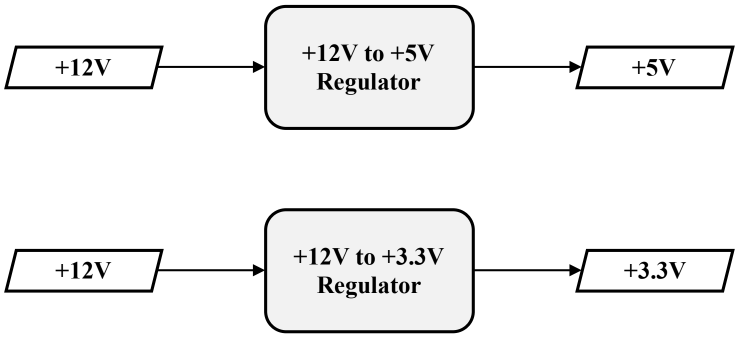


Figure 11: Level 2 Power Supply Block Diagram

Module	Level 2: DC/DC Regulators
Designers	John Martaus, Elliott Boudreau
Inputs	Power Supply: 12V
Outputs	Power Output: 3.3V and 5V
Description	The regulators supply 3.3V and 5V to the control circuitry with a supply voltage of 12V.

Table 8: Level 2: DC/DC Regulators Functional Requirements

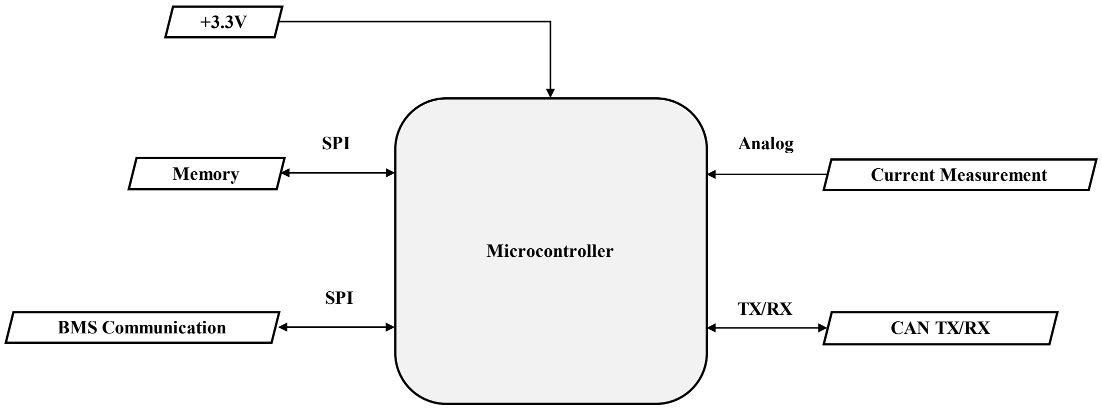


Figure 12: Level 2 Microcontroller Hardware Block Diagram

Module	Level 2: Microcontroller
Designers	John Martaus, Elliott Boudreau
Inputs	Data: SPI Data received from sense boards, memory, transceivers, current sensor Power Supply: 3.3V
Outputs	Control Data: SPI data sent to the sense boards, memory, transceivers CAN Signals: 1MBaud
Description	The microcontroller sends and receives all data for the BMS.

Table 9: Level 2 Microcontroller Functional Requirements

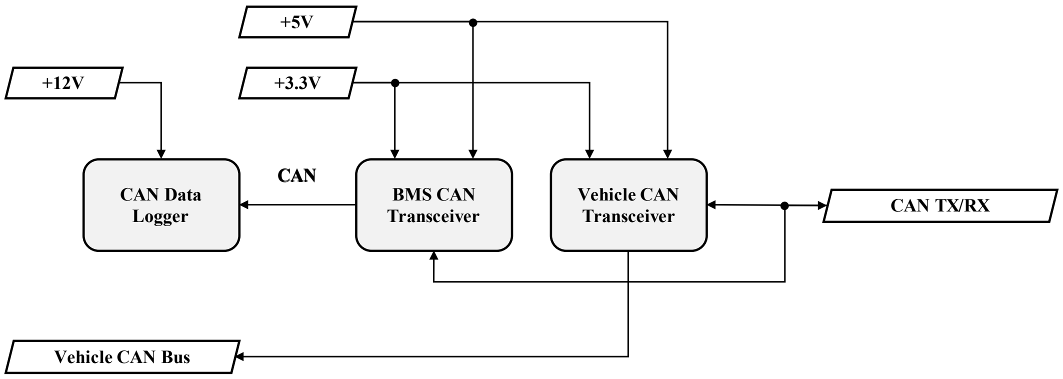


Figure 13: Level 2 CAN Bus Hardware Block Diagram

Module	Level 2: CAN Transceivers
Designers	John Martaus, Elliott Boudreau
Inputs	Data: TX/RX signals from the microcontroller Power supply: 3.3V and 5V
Outputs	Data: CAN data sent to internal CAN bus and vehicle CAN bus
Description	The CAN transceivers are used to take the data from the microcontroller and convert it to CAN signals that can be read by the devices of the vehicle CAN and recorded on the datalogger.

Table 10: Level 2 CAN Transceivers Functional Requirements

Module	Level 2: CAN Data Logger
Designers	John Martaus, Elliott Boudreau
Inputs	Data: CAN signals from transceiver. Power supply: 12V
Description	The CAN datalogger records all CAN messages so that data may be analyzed after driving

Table 11: Level 2 CAN Data Logger Functional Requirements

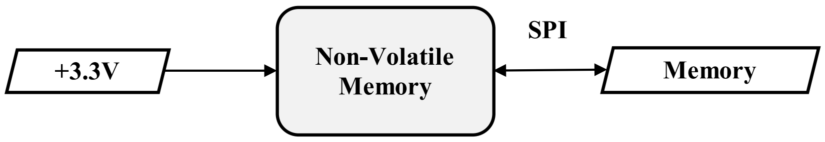


Figure 14: Level 2 Memory Hardware Block Diagram

Module	Level 2: Non-Volatile Memory
Designers	John Martaus, Elliott Boudreau
Inputs	Power Supply: 3.3V Data: Sent via SPI from the microcontroller
Outputs	Data: Sent via SPI to microcontroller when requested
Description	The non-volatile memory is used to store data safely during power-down situations.

Table 12: Level 2 Memory Functional Requirements

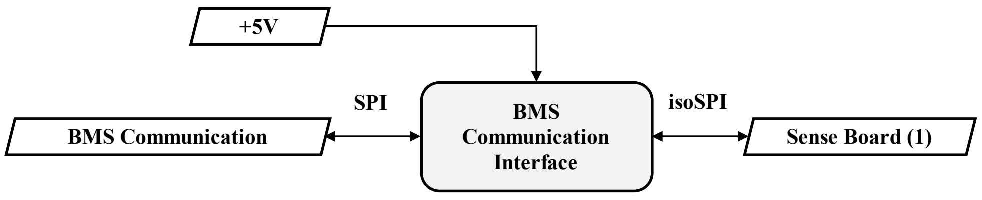


Figure 15: Level 2 BMS Communication Interface Hardware Block Diagram

Module	Level 2: BMS Communication Interface
Designers	John Martaus, Elliott Boudreau
Inputs	Power Supply: 5V Data: Sent via SPI from the microcontroller and sent via isoSPI from the sense boards
Outputs	Data: Sent via SPI to microcontroller when requested and sent via isoSPI to the sense boards
Description	The BMS communication interface is used to convert SPI data to isoSPI data or vice versa to allow for isolated communication between the sense boards and the control board.

Table 13: Level 2 BMS Communication Interface Function Requirements

The control board from level 1 is split into the required sections for the level 2 block diagram. The control board houses the DC/DC regulators, the microcontroller, the CAN transceivers, the memory, and the BMS communication interface circuitry.

The board contains two DC/DC regulators that step down the LV battery voltage (Nominal Voltage: +12V) to both +5V and +3.3V which is used to power the various circuitry housed on the board. The microcontroller communicates via SPI with multiple sections of the block diagrams including the memory, CAN transceivers, and the BMS communication interface. The memory is used to store critical data that must be saved during power down situations. There are two CAN transceivers on the control board. One is used for internal BMS CAN bus and the other is used to send critical data to the vehicle CAN bus. The BMS communication interface is a SPI to isoSPI transceiver which enables communication between the micro controller and the sense boards.

Author: JM

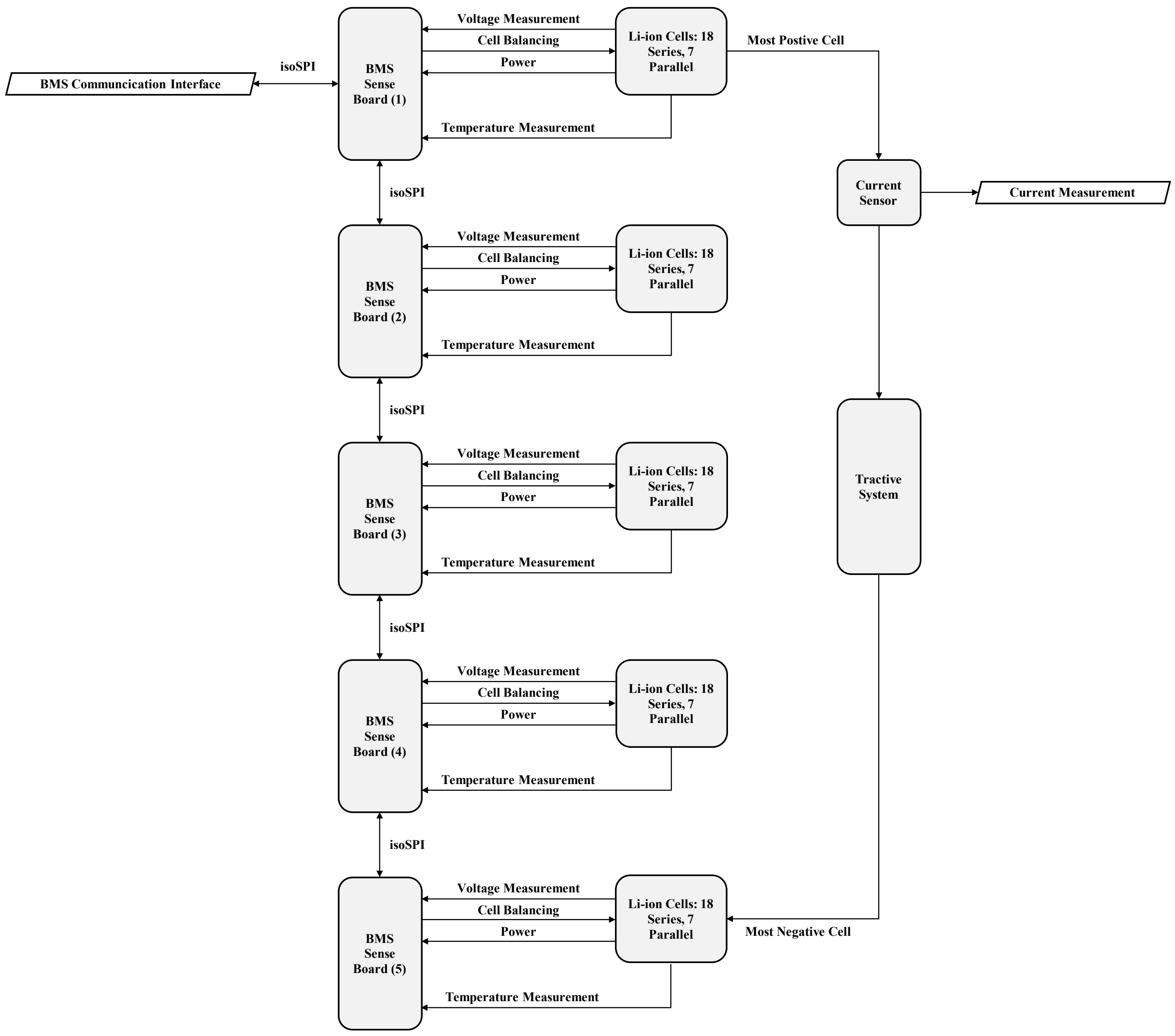


Figure 16: Level 2 Sensing Board Hardware Block Diagram

Module	Level 2: BMS Sense boards
Designers	John Martaus, Elliott Boudreau
Inputs	Power Supply: Powered by batteries it monitors Cell temperature: Measures 20% of all cells. Max 60˚C Cell Voltage: Measures voltage of all the cells. Data: Sends data from the microcontroller via isoSPI.
Outputs	Voltage Balancing: Balances all cells. Data: Controlled by commands sent from the microcontroller via isoSPI
Description	Reads temperature and voltages and relays data to the microcontroller. Balances all cells.

Table 14: Level 2 BMS Sense Boards Functional Requirements

Module	Level 2: Current Sensor
Designers	John Martaus, Elliott Boudreau
Outputs	Current measurement: Analog measurement sent directly to microcontoller.
Description	Measures the tractive system current. Used for Coulomb counting.

Table 15: Level 2 Current Senor Functional Requirements

Module	Level 2: Li-on Cells
Designers	John Martaus, Elliott Boudreau
Inputs	Sensing: Voltage sense taps to measure cell voltage. Power: Powers the BMS Sense Boards.
Outputs	Tractive system voltage and current used to spin motor.
Description	The Li-ion cells are used to power the tractive system. There are 18 cells in series and 7 in parallel.

Table 16: Level 2 Li-ion Cells Functional Requirements

Module	Level 2: Tractive System
Designers	John Martaus, Elliott Boudreau
Inputs	Power: Tractive system voltage and current used to spin motor.
Outputs	Torque output to powertrain
Description	The tractive system includes the motor controller/inverter and the motor. The tractive system is power by the configuration of li-ion battery cells.

Table 17: Level 2 Tractive System Functional Requirements

The sense boards communicate with the control board via isoSPI. Commands are sent from the microcontroller and data from the sense boards is sent back to the microcontroller. The sense boards monitor the voltage of all the 18 cells in series and in each of the 5 segments. The temperature of 20% of the cells is monitored by the sense boards. The temperature sensors must be evenly distributed in all the segments. The sense boards also balance the cells via the voltage sense tap lines. The tractive system is powered by the battery segments. The overall tractive system current is measured with a current sensor and the data is sent directly to the microcontroller where it is used for Coulomb counting for the state of charge calculations.

Author: JM

Hardware Circuit Schematics, Models, and Assembly – Accumulator Management System (Control Board)

*Note: The complete AMS schematics have been provided. Only the relevant BMS schematic sheets will be discussed.

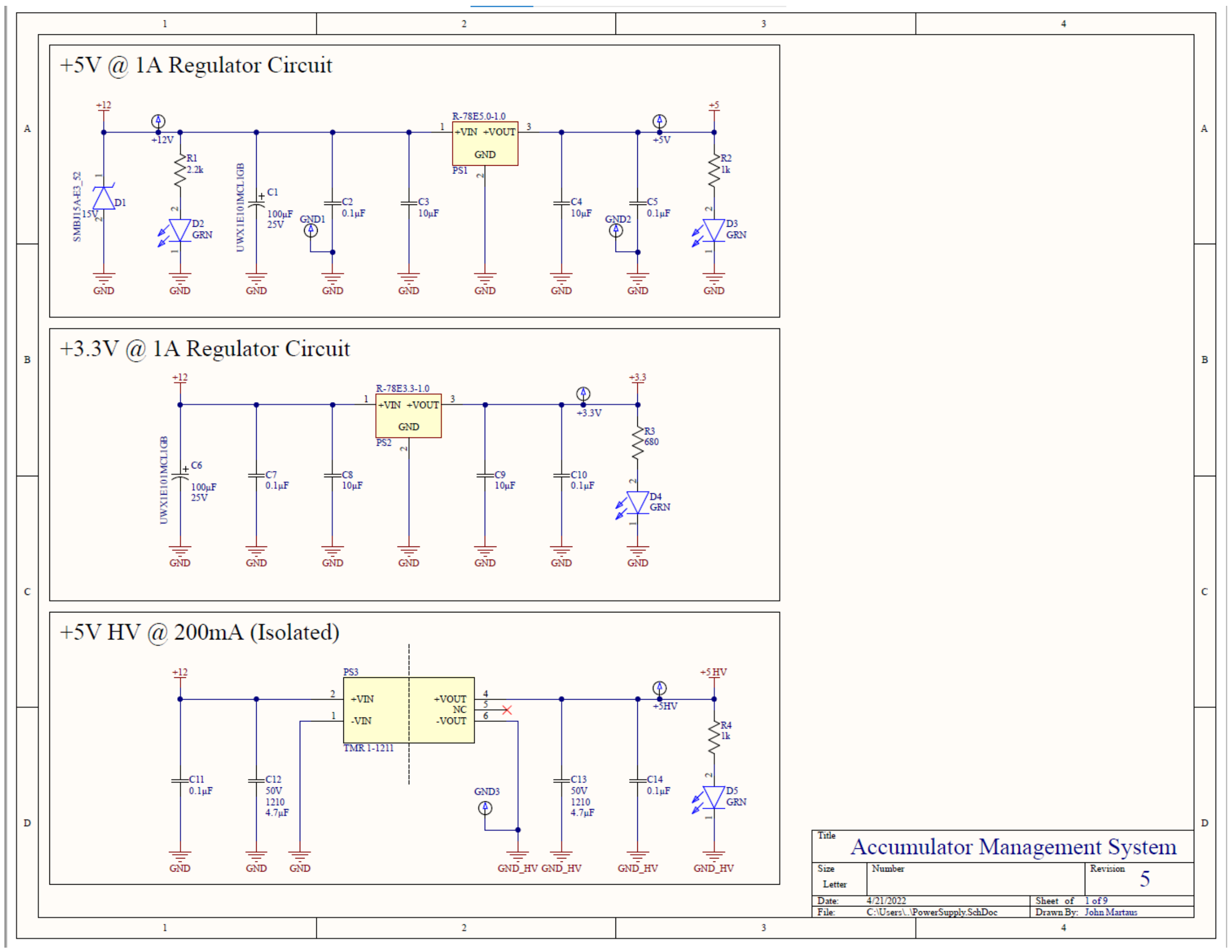


Figure 17: AMS Schematics - 1

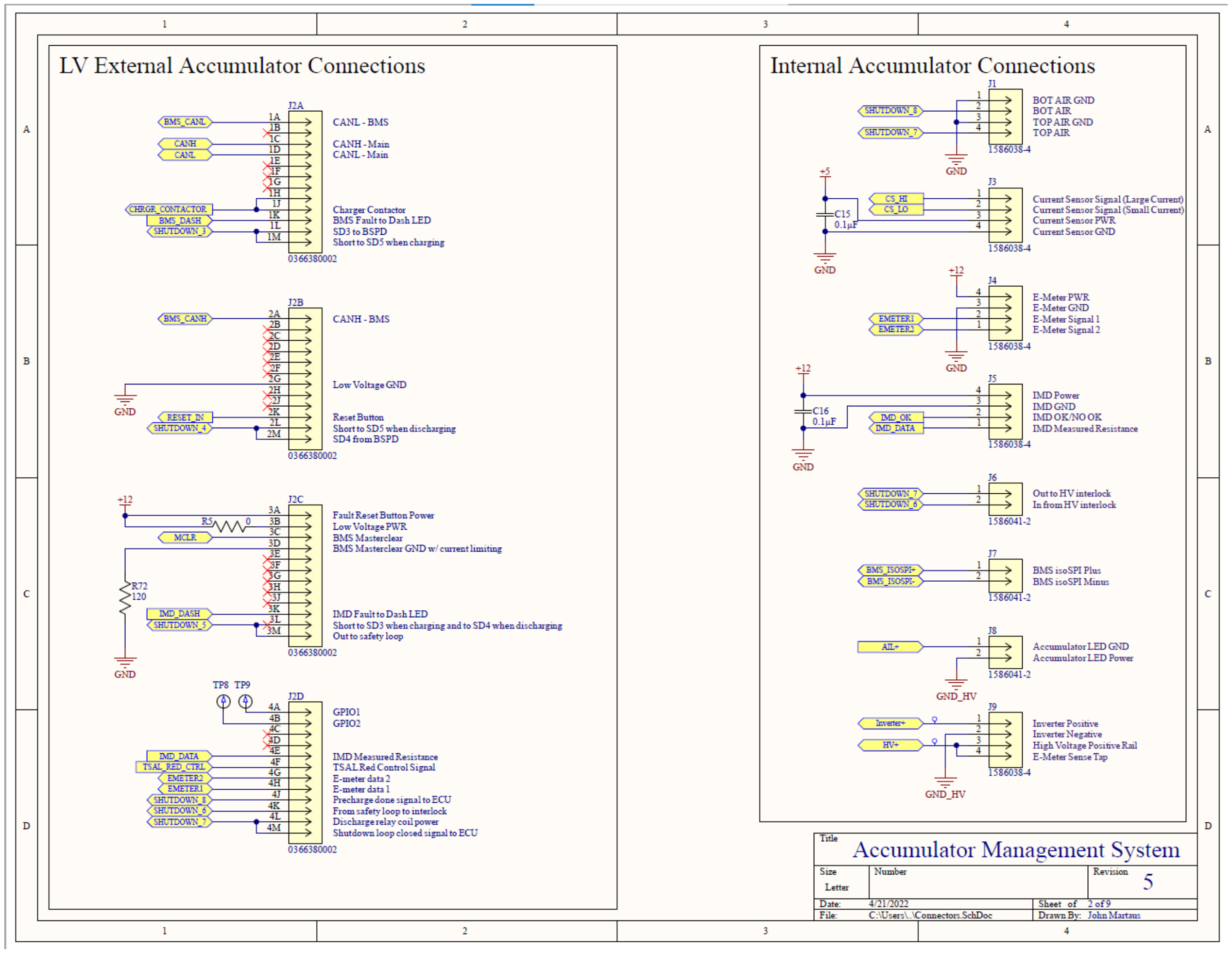


Figure 18: AMS Schematics - 2

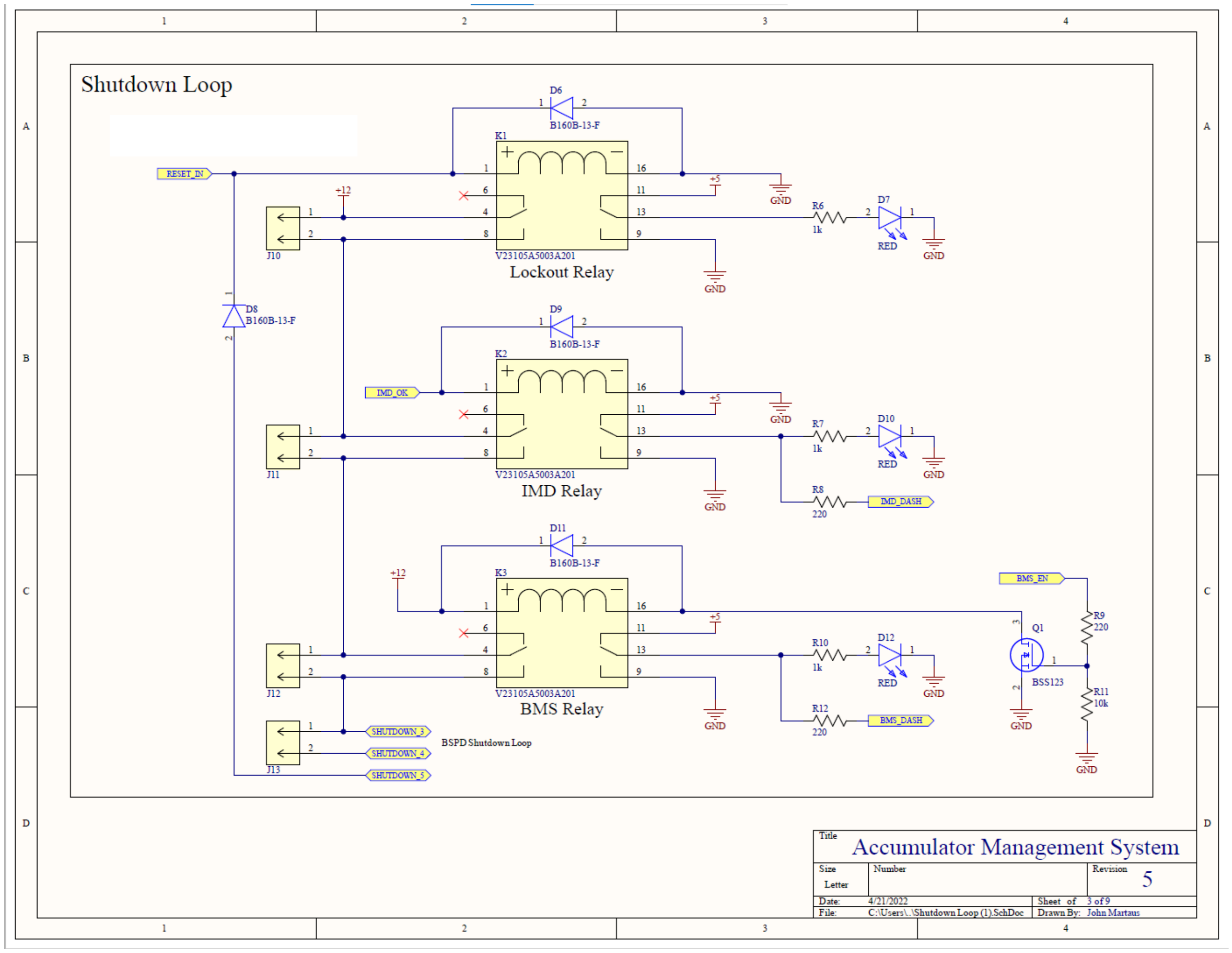


Figure 19: AMS Schematics - 3

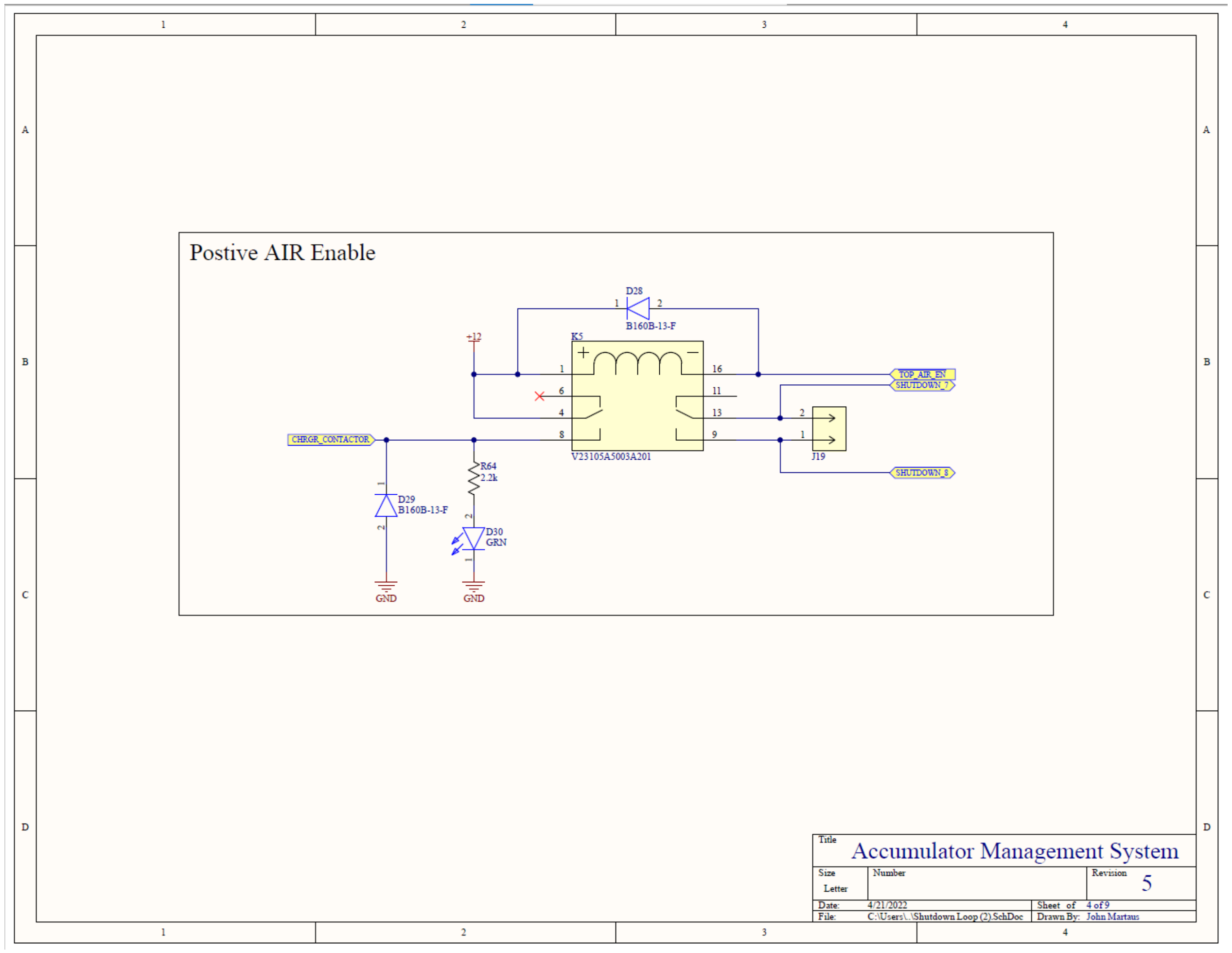


Figure 20: AMS Schematics - 4

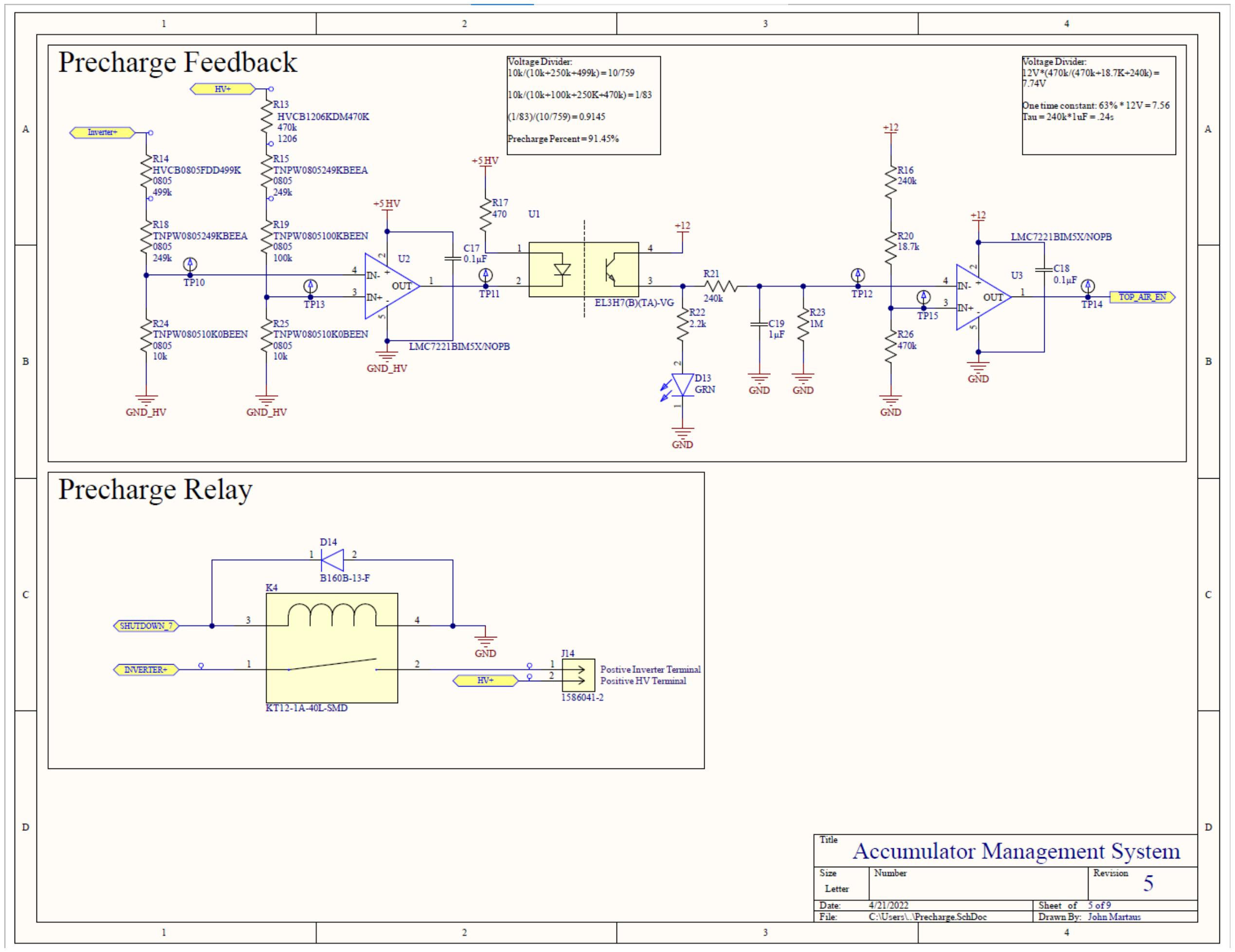


Figure 21: AMS Schematics - 5

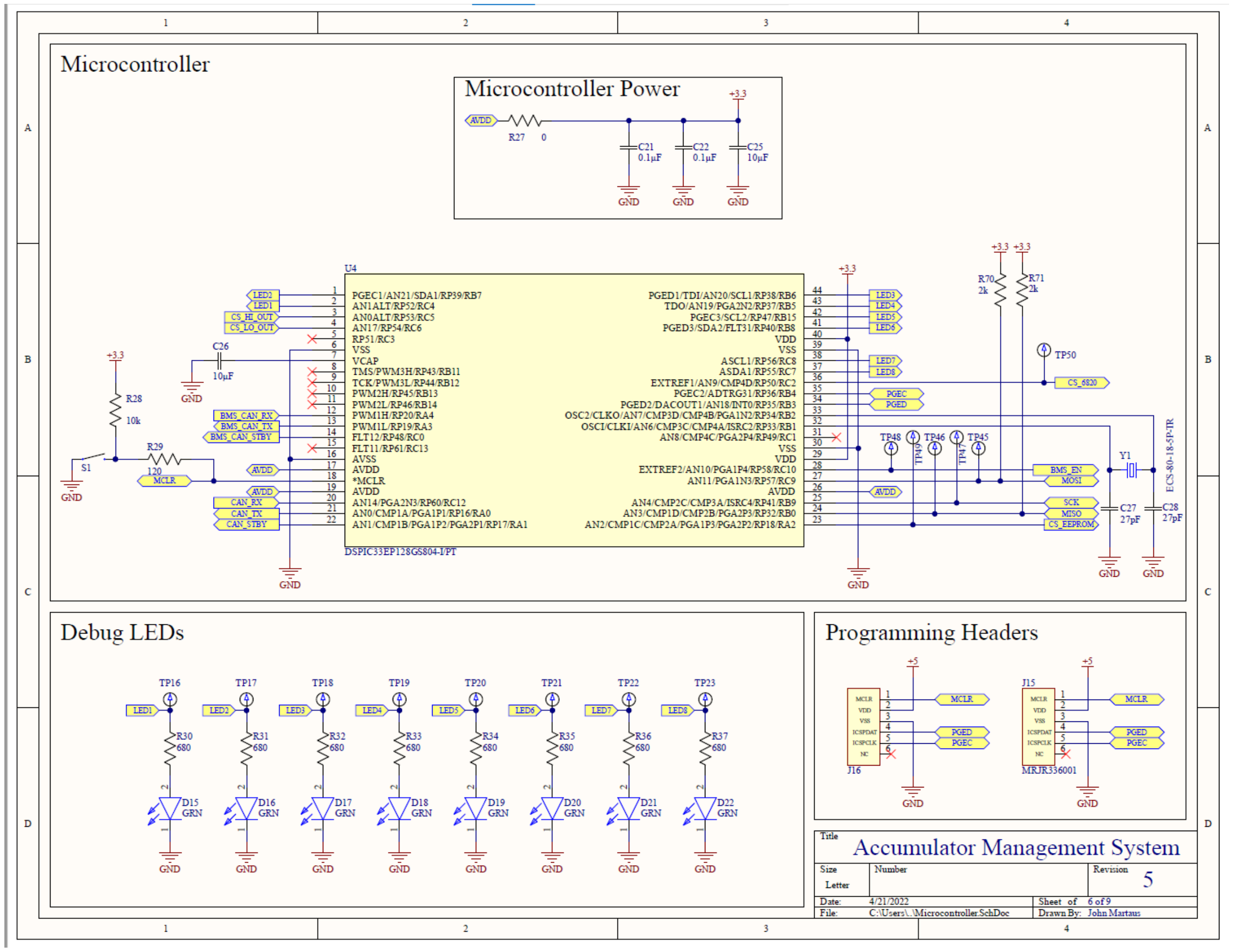


Figure 22: AMS Schematics - 6

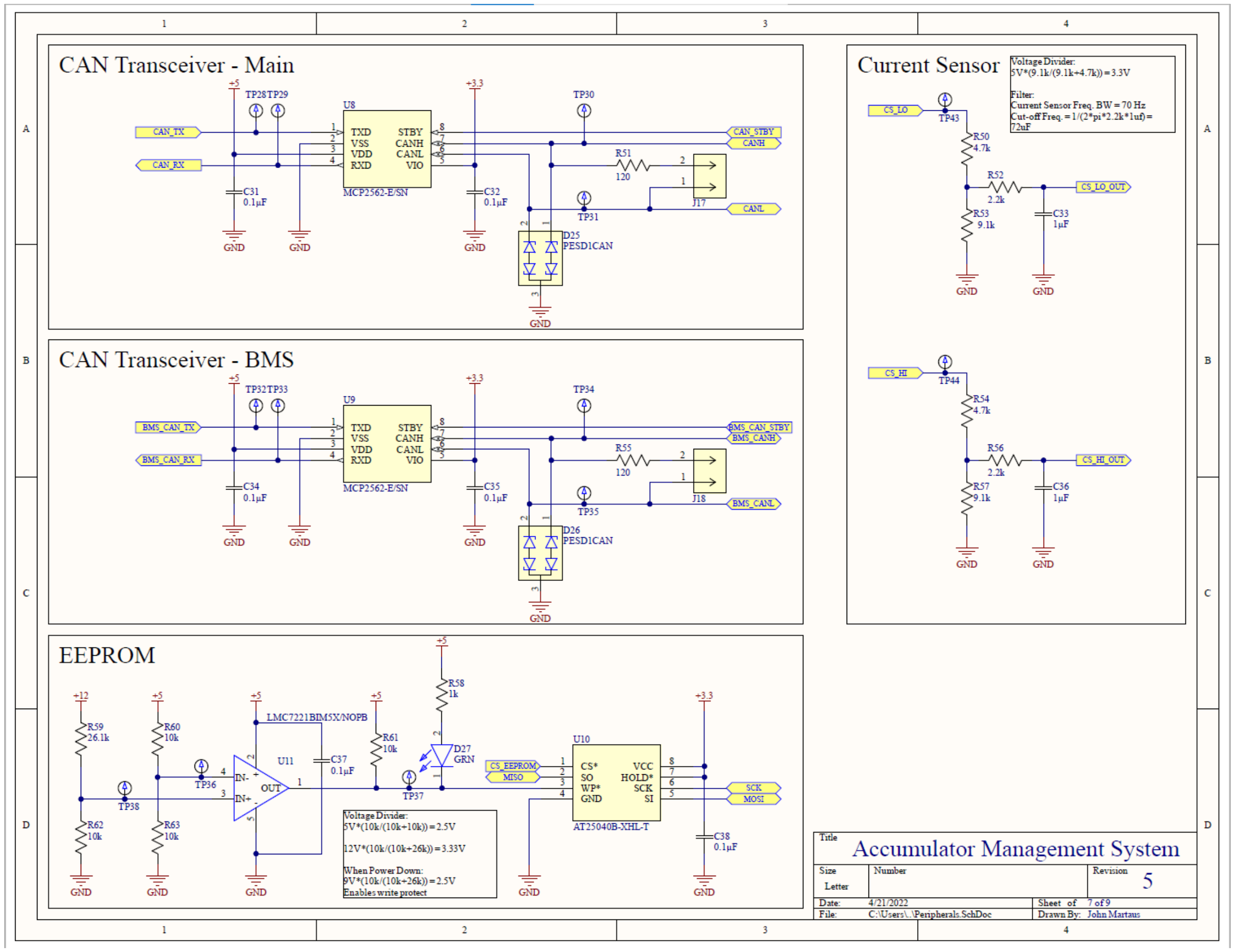


Figure 23: AMS Schematics - 7

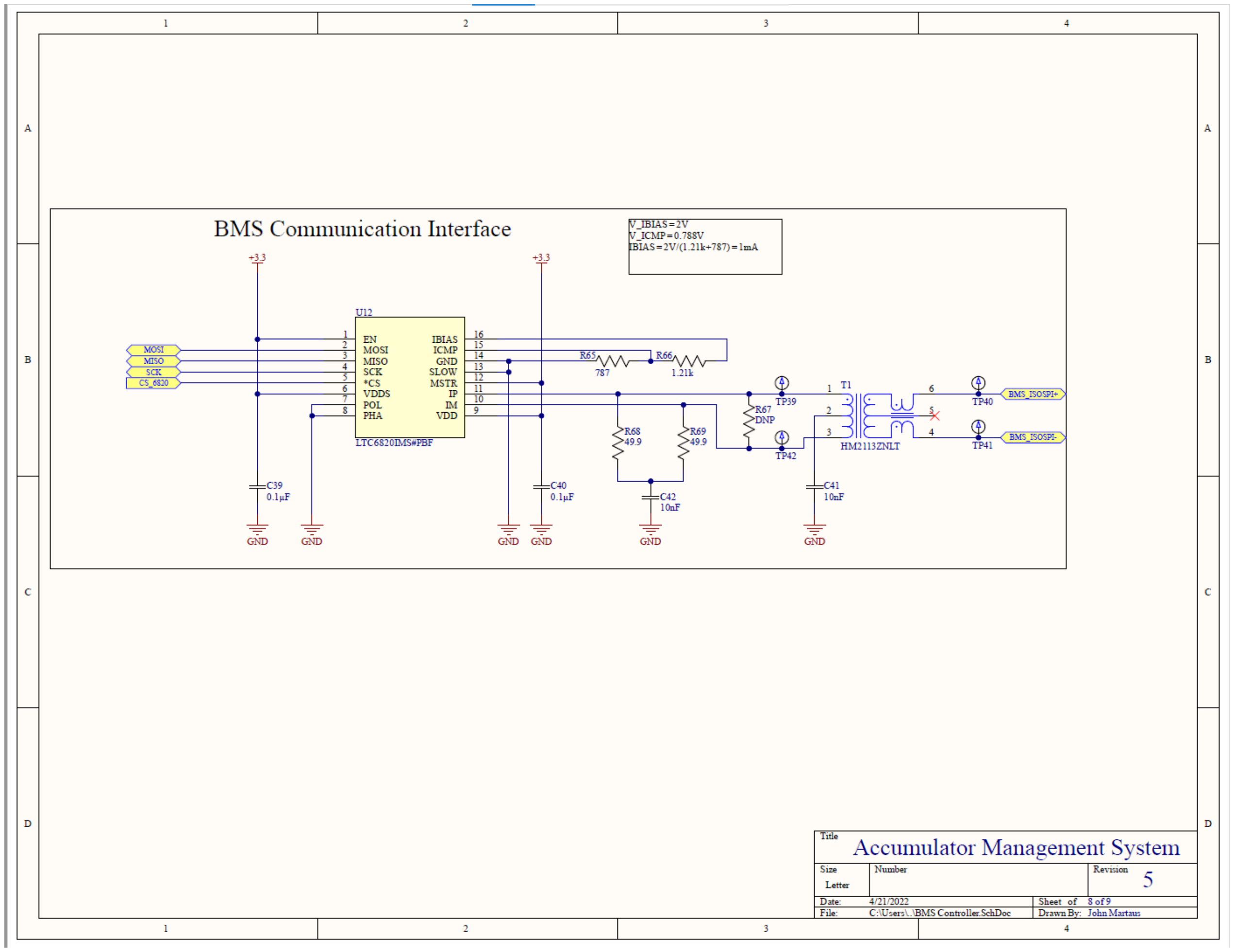


Figure 24: AMS Schematics - 8

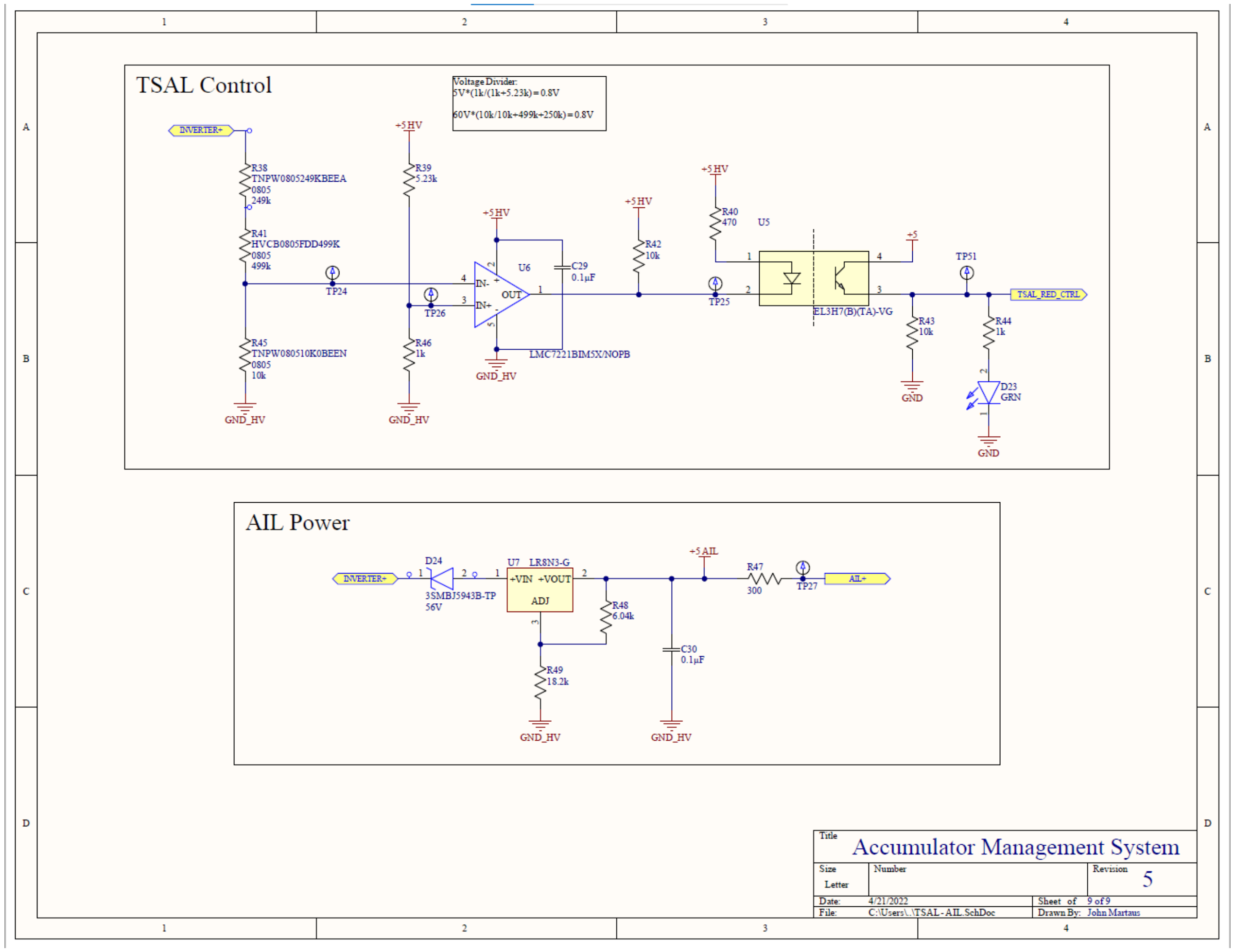


Figure 25: AMS Schematics – 9

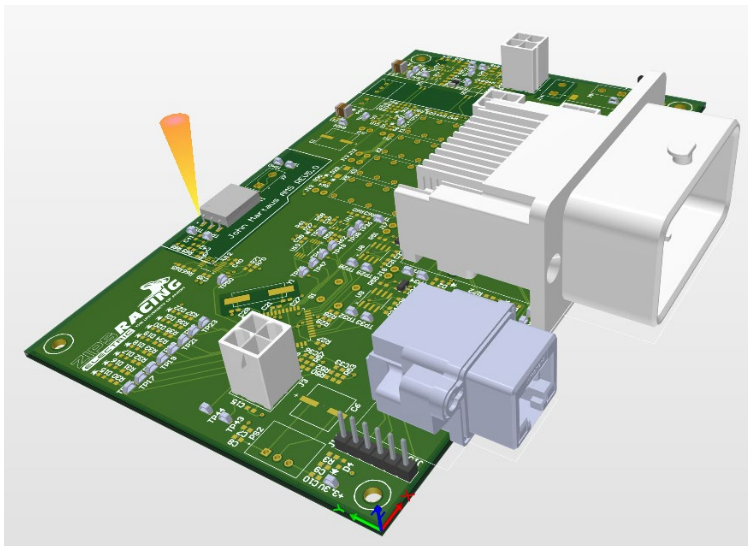


Figure 26: AMS - 3D Model

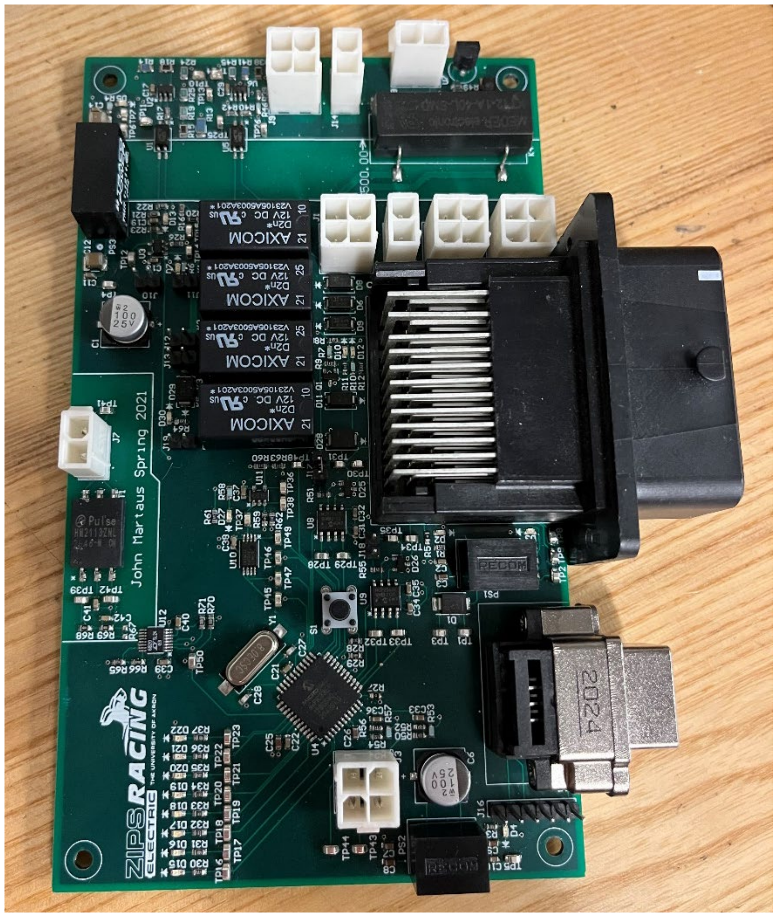


Figure 27: AMS - Assembled PCB

Hardware Schematics Discussion – Accumulator Management System (Control Board)

Sheet 1:

The DC/DC regulators step +12V down to +5V and +3.3V. A diode overvoltage protection device rated for +15V is placed across the input. The circuit includes tank capacitors for large load spikes and filter capacitors to suppress high frequency noise. LEDs in parallel with the regulators show that +12V and the respective stepped-down voltage is being outputted.

Sheet 2:

The current sensor is connected to the schematic through a 4-pin connector. Power and ground are supplied, and the analog data is received through the connector.

Sheet 3:

The BMS enable is used to close the shutdown loop of the car which allows the car to turn on. If there is any fault within the BMS, the enable pin is pulled low which opens the shutdown loop and shuts down the car.

Sheet 6:

The microcontroller is setup to the specifications set in the datasheet. Capacitors are placed across the supply pins for decoupling. A master clear switch is included with a pull up resistor and current limiting resistor. The microcontroller communicates with the EEPROM, the CAN transceivers, and the BMS interface via SPI. The microcontroller is programmed with either the Microchip PICKit or the ICD4 through the programming header and connector provided. Software debug LEDs are included to assist with software testing.

Sheet 7:

The CAN transceivers are powered with +5V and +3.3V is used to set the digital threshold for the CAN bus. Transient voltage suppressors are included to filter noise on the CAN bus. The current sensor inputs, which have 5V logic levels, are stepped down to 3.3V, so the microcontroller is able to read the entire data output.

Sheet 8:

The BMS communication interface receives SPI signals and outputs the isoSPI signals. A transformer is included to galvanically isolate the control board from the sense boards in accordance with the SAE rules.

Author: JM

Accumulator Management System (Control Board) – Bill of Materials

Comment	Description	Designator
TP5015	PC TEST POINT MINIATURE	+3.3V, +5HV, +5V, +12V, GND1, GND2, GND3, TP8, TP9, TP10, TP11, TP12, TP13, TP14, TP15, TP16, TP17, TP18, TP19, TP20, TP21, TP22, TP23, TP24, TP25, TP26, TP27, TP28, TP29, TP30, TP31, TP32, TP33, TP34, TP35, TP36, TP37, TP38, TP39, TP40, TP41, TP42, TP43, TP44, TP45, TP46, TP47, TP48, TP49, TP50, TP51
UWX1E101MCL1GB	100 µF 25 V Aluminum Electrolytic Capacitors Radial, Can - SMD - 2000 Hrs @ 85°C	C1, C6
CAP_0603_0.1µF	0.1µF Ceramic Capacitor 0603	C2, C5, C7, C10, C11, C14, C15, C16, C17, C18, C21, C22, C29, C30, C31, C32, C34, C35, C37, C38, C39, C40
CAP_0603_10µF	10µF Ceramic Capacitor 0603	C3, C4, C8, C9, C25, C26
CAP_1210_4.7µF	4.7 µF ±10% 50V Ceramic Capacitor X7R 1210 (3225 Metric)	C12, C13
CAP_0603_1µF	1µF Ceramic Capacitor 0603	C19, C33, C36
CAP_0603_27pF	27pF Ceramic Capacitor 0603	C27, C28
CAP_0603_10nF	10nF Ceramic Capacitor 0603	C41, C42
SMBJ15A-E3_52	TVS DIODE 15V 24.4V DO214AA	D1
GRN	Generic - Green	D2, D3, D4, D5, D13, D15, D16, D17, D18, D19, D20, D21, D22, D23, D27, D30
B160B-13-F	DIODE SCHOTTKY 60V 1A SMA	D6, D8, D9, D11, D14, D28, D29
RED	Generic - Red	D7, D10, D12
3SMBJ5943B-TP	Zener Diode 56 V 3 W ±5% Surface Mount DO-214AA (SMB)	D24
PESD1CAN	50V Clamp 3A (8/20µs) Ipp Tvs Diode Surface Mount SOT-323	D25, D26
1586038-4	Connector Header Through Hole 4 position 0.165" (4.20mm)	J1, J3, J4, J5, J9
0366380002	Connector Header Board Edge, Through Hole, Right Angle 48 position	J2
1586041-2	Connector Header Through Hole 2 position 0.165" (4.20mm)	J6, J7, J8, J14
JMPR	2 Pin Jumper	J10, J11, J12, J13, J17, J18, J19
MRJR336001	Jack Modular Connector 6p6c (RJ11, RJ12, RJ14, RJ25) 90° Angle (Right) Unshielded	J15
M20-9990646	Connector Header Through Hole 6 position 0.100" (2.54mm)	J16
V23105A5003A201	RELAY TELECOM DPDT 3A 12V	K1, K2, K3, K5
KT12-1A-40L-SMD	REED RELAY 1 FORM A 12V SMD	K4
R-78E5.0-1.0	Linear Regulator Replacement DC DC Converter 1 Output 5V - - 1A 8V - 28V Input	PS1
R-78E3.3-1.0	Linear Regulator Replacement DC DC Converter 1 Output 3.3V - - 1A 7V - 28V Input	PS2
TMR 1-1211	Isolated Module DC DC Converter 1 Output 5V - - - 200mA 9V - 18V Input	PS3
BSS123	MOSFET N-CH 100V 170MA SOT23-3	Q1
RES_0603_2.2k	RES SMD 2.2k OHM 1% 0603	R1, R22, R52, R56, R64
RES_0603_1k	RES SMD 1K OHM 1% 0603	R2, R4, R6, R7, R10, R44, R46, R58
RES_0603_680	RES SMD 680 OHM 1% 0603	R3, R30, R31, R32, R33, R34, R35, R36, R37
RES_0603_0_OHM	RES SMD 0 OHM 1% 0603	R5, R27
RES_0603_220	RES SMD 220 OHM 1% 0603	R8, R9, R12
RES_0603_10k	RES SMD 10K OHM 1% 0603	R11, R28, R42, R43, R60, R61, R62, R63
HVCB1206KDM470K	470 kOhms ±10% 0.333W, 1/3W Chip Resistor 1206 (3216 Metric) High Voltage, Moisture Resistant, Pulse Withstanding Thick Film	R13
HVCB0805FDD499K	499k ±1% 0.2W, 1/5W Chip Resistor 0805 (2012 Metric) High Voltage, Moisture Resistant, Pulse Withstanding Thick Film	R14, R41
TNPW0805249KBEEA	249 kOhms ±0.1% 0.2W, 1/5W Chip Resistor 0805 (2012 Metric) Anti-Sulfur, Automotive AEC-Q200, Moisture Resistant Thin Film	R15, R18, R38
RES_0603_240k	RES SMD 240k OHM 1% 0603	R16, R21
RES_0603_470	RES SMD 470 OHM 1% 0603	R17, R40
TNPW0805100KBEEN	100 kOhms ±0.1% 0.2W, 1/5W Chip Resistor 0805 (2012 Metric) Anti-Sulfur, Automotive AEC-Q200, Moisture Resistant Thin Film	R19
RES_0603_18.7k	RES SMD 18.7K OHM 1% 0603	R20
RES_0603_1M	RES SMD 1M OHM 1% 0603	R23
TNPW080510K0BEEN	10 kOhms ±0.1% 0.2W, 1/5W Chip Resistor 0805 (2012 Metric) Anti-Sulfur, Automotive AEC-Q200, Moisture Resistant Thin Film	R24, R25, R45
RES_0603_470k	RES SMD 470K OHM 1% 0603	R26
RES_0603_120	RES SMD 120 OHM 1% 0603	R29, R51, R55, R72
RES_0603_5.23k	RES SMD 5.23 kOHM 1% 0603	R39
RES_0603_300	RES SMD 300 OHM 1% 0603	R47
RES_0603_6.04k	RES SMD 6.04K OHM 1% 0603 (RCS06036K04FKEA)	R48
RES_0603_18.2k	RES SMD 18.2K OHM 1% 0603 (RCS060318K2FKEA)	R49
RES_0603_4.7k	RES SMD 4.7k OHM 1% 0603	R50, R54
RES_0603_9.1k	RES SMD 9.1k OHM 1% 0603	R53, R57
RES_0603_26.1k	RES SMD 26.1k OHM 1% 0603	R59
RES_0603_787	RES SMD 787 OHM 1% 0603	R65
RES_0603_1.21k	RES SMD 1.21k OHM 1% 0603	R66
RES_0603_DNP	RES SMD DNP 0603	R67
RES_0603_49.9	RES SMD 49.9 OHM 1% 0603	R68, R69
RES_0603_2k	RES SMD 2K OHM 1% 0603	R70, R71
3-1825910-5	Tactile Switch	S1
HM2113ZNLT	150µH, 450µH - Pulse Transformer 1:1 Surface Mount	T1
EL3H7(B)(TA)-VG	Optoisolator Transistor Output 3750Vrms 1 Channel 4-SSOP, No Description Available	U1, U5
LMC7221BIM5X/NOPB	Comparator General Purpose Open Drain SOT-23-5	U2, U3, U6, U11
DSPIC33EP128GS804-I/PT	dsPIC series Microcontroller IC 16-Bit 70 MIPs 128KB (43K x 24) FLASH 44-TQFP (10x10)	U4
LR8N3-G	Linear Voltage Regulator IC Positive Adjustable 1 Output 10mA TO-92-3	U7
MCP2562-E/SN	1/1 Transceiver Half CANbus 8-SOIC	U8, U9
AT25040B-XHL-T	EEPROM Memory IC 4Kb (512 x 8) SPI 20 MHz 8-TSSOP	U10
LTC6820IMS#PBF	Isolated Communications Interface Interface 16-MSOP	U12
ECS-80-18-5P-TR	8MHz ±30ppm Crystal 18pF 60 Ohms HC-49/US	Y1

Table 18: AMS Bill of Materials

Accumulator Management System (Control Board) – Materials Budget

Qty.	Refdes	Part Num.	Description	Suggested Vendor	Vendor Part Num.	Cost	Total Cost
30	D6, D8, D9, D11,	B160B-13-F	DIODE SCHOTTKY 60V 1A SMA	Digikey	B160B-FDITR-ND	$ 0.74	$ 22.20
10	D24	1SMA4758-GT3TR	Zener Diode 56 V 1 W ±5% Surface Mount SMA (DO-214AC)	Digikey	1655-2305-2-ND	$ 0.42	$ 4.20
15	K1, K2, K3, K5	V23105A5003A201	RELAY TELECOM DPDT 3A 12V	Digikey	PB384-ND	$ 3.76	$ 56.40
5	PS3	TMR 1-1211	Isolated Module DC DC Converter 1 Output 5V - - - 200mA 9V - 18V Input	Digikey	1951-2675-ND	$ 9.20	$ 46.00
10	U7	LR8N3-G	Linear Voltage Regulator IC Positive Adjustable 1 Output 10mA TO-92-3	Digikey	LR8N3-G-ND	$ 0.72	$ 7.20
10	C1, C6	UWX1E560MCL1GB	CAP ALUM 56UF 20% 25V SMD	Digikey	493-2112-2-ND	$ 0.41	$ 4.10
20	R15, R18, R38	TNPW0805249KBEEA	249 kOhms ±0.1% 0.2W, 1/5W Chip Resistor 0805 (2012 Metric) Anti-Sulfur, Automotive AEC-Q200, Moisture Resistant Thin Film	Digikey	TNP249KABTR-ND	$ 0.78	$ 15.60
20	R19	TNPW0805100KBEEN	100 kOhms ±0.1% 0.2W, 1/5W Chip Resistor 0805 (2012 Metric) Anti-Sulfur, Automotive AEC-Q200, Moisture Resistant Thin Film	Digikey	541-3053-2-ND	$ 0.92	$ 18.40
10	R24, R25, R45	TNPW080510K0BEEN	10 kOhms ±0.1% 0.2W, 1/5W Chip Resistor 0805 (2012 Metric) Anti-Sulfur, Automotive AEC-Q200, Moisture Resistant Thin Film	Digikey	541-3723-2-ND	$ 0.92	$ 9.20
10	U1	EL3H7(TA)-VG	Optoisolator Transistor Output 3750Vrms 1 Channel 4-SSOP	Digikey	1080-1198-2-ND	$ 0.49	$ 4.90
10	U10	AT25040B-XHL-T	EEPROM Memory IC 4Kb (512 x 8) SPI 20 MHz 8-TSSOP	Digikey	AT25040B-XHL-TTR-ND	$ 0.32	$ 3.20
10	R13	HVCB1206KDM470K	470 kOhms ±10% 0.333W, 1/3W Chip Resistor 1206 (3216 Metric) High Voltage, Moisture Resistant, Pulse Withstanding Thick Film	Digikey	HVCB1206KDM470KTR-ND	$ 2.67	$ 26.70
10	R14, R41	HVCB0805FDD499K	499k ±1% 0.2W, 1/5W Chip Resistor 0805 (2012 Metric) High Voltage, Moisture Resistant, Pulse Withstanding Thick Film	Digikey	HVCB0805FDD499KTR-ND	$ 5.83	$ 58.30
2	J2	0366380002	Connector Header Board Edge, Through Hole, Right Angle 48 position	Digikey	WM4438-ND	$ 11.73	$ 23.46
25	D2, D3, D4, D5, D	LTST-C190TBKT	LED BLUE CLEAR CHIP SMD	Digikey	160-1646-2-ND	$ 0.35	$ 8.75
30	D7, D10, D12	LTST-C190KFKT	LED ORANGE CLEAR CHIP SMD	Digikey	160-1434-1-ND	$ 0.23	$ 6.90
25	U2, U3, U11	LMC7221BIM5	IC COMPAR TINY R-R CMOS SOT23-5	Digikey	296-LMC7221BIM5TR-ND	$ 2.39	$ 59.75
5	J15	MRJR336001	Jack Modular Connector 6p6c (RJ11, RJ12, RJ14, RJ25) 90° Angle (Right) Unshielded	Mouser	523-MRJR-3360-01	$ 13.09	$ 65.45

Table 19: AMS Materials Budget

Hardware Circuit Schematics, Models, and Assembly – Battery Management System (Sense and Discharge Board)

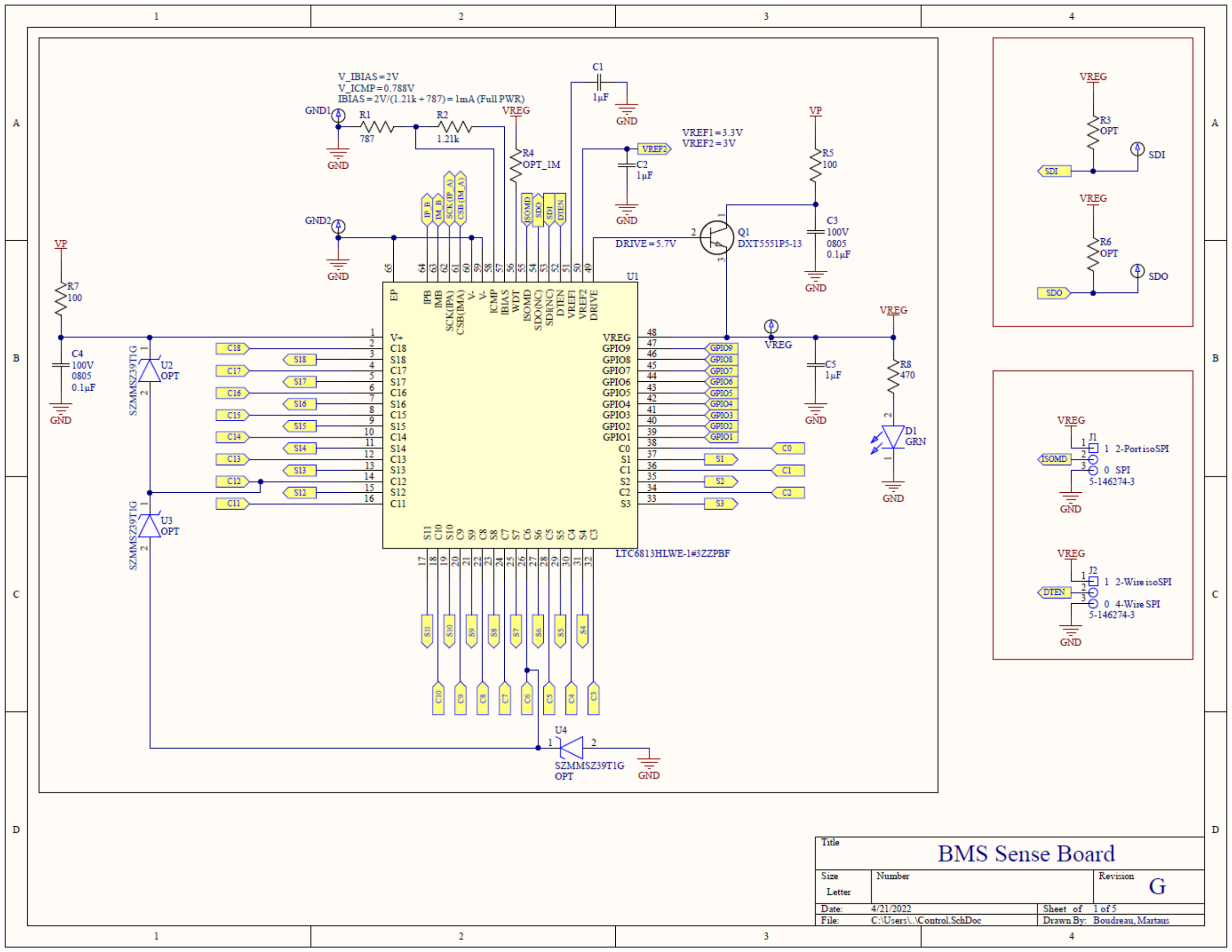


Figure 28: BMS Schematics – 1

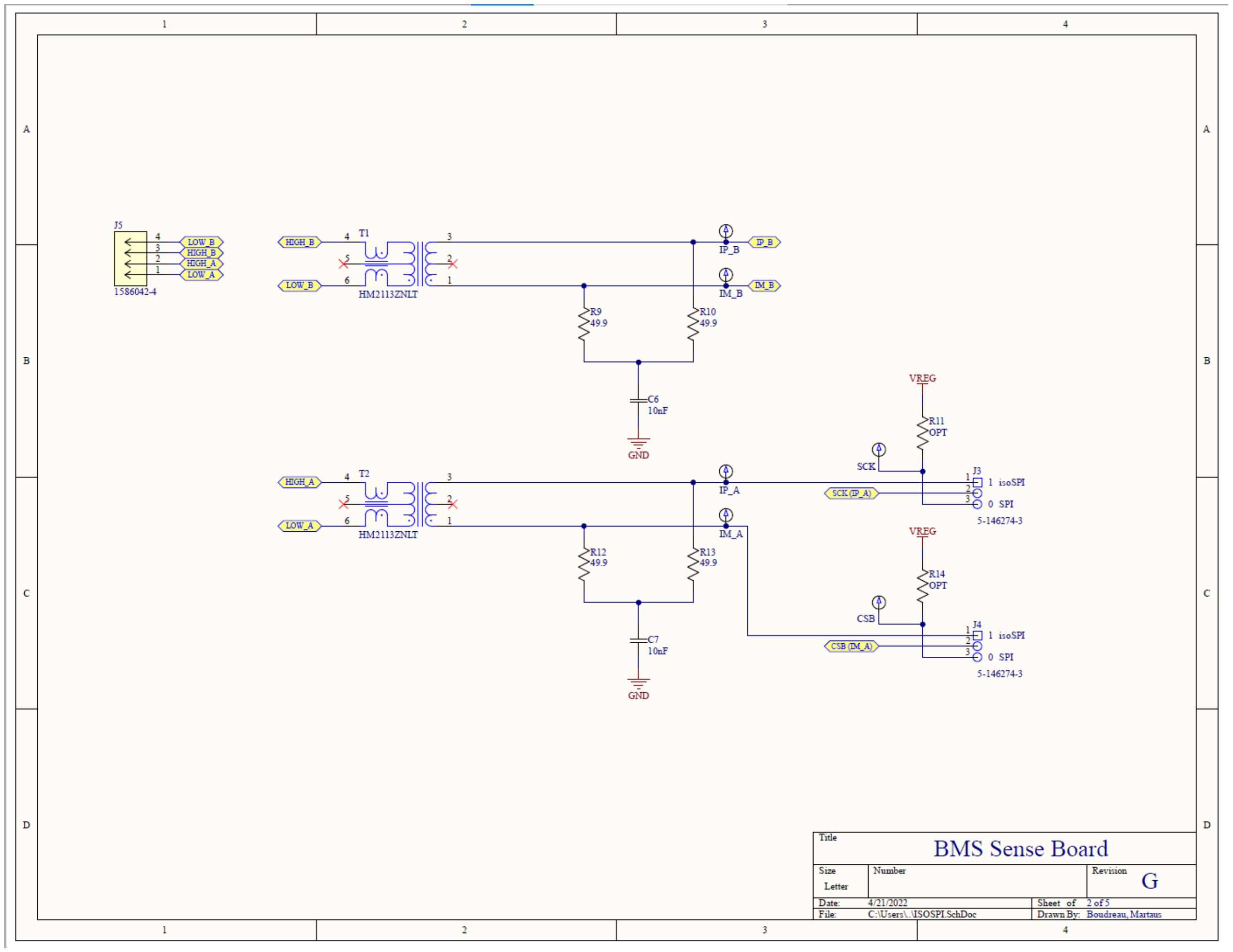


Figure 29: BMS Schematics – 2

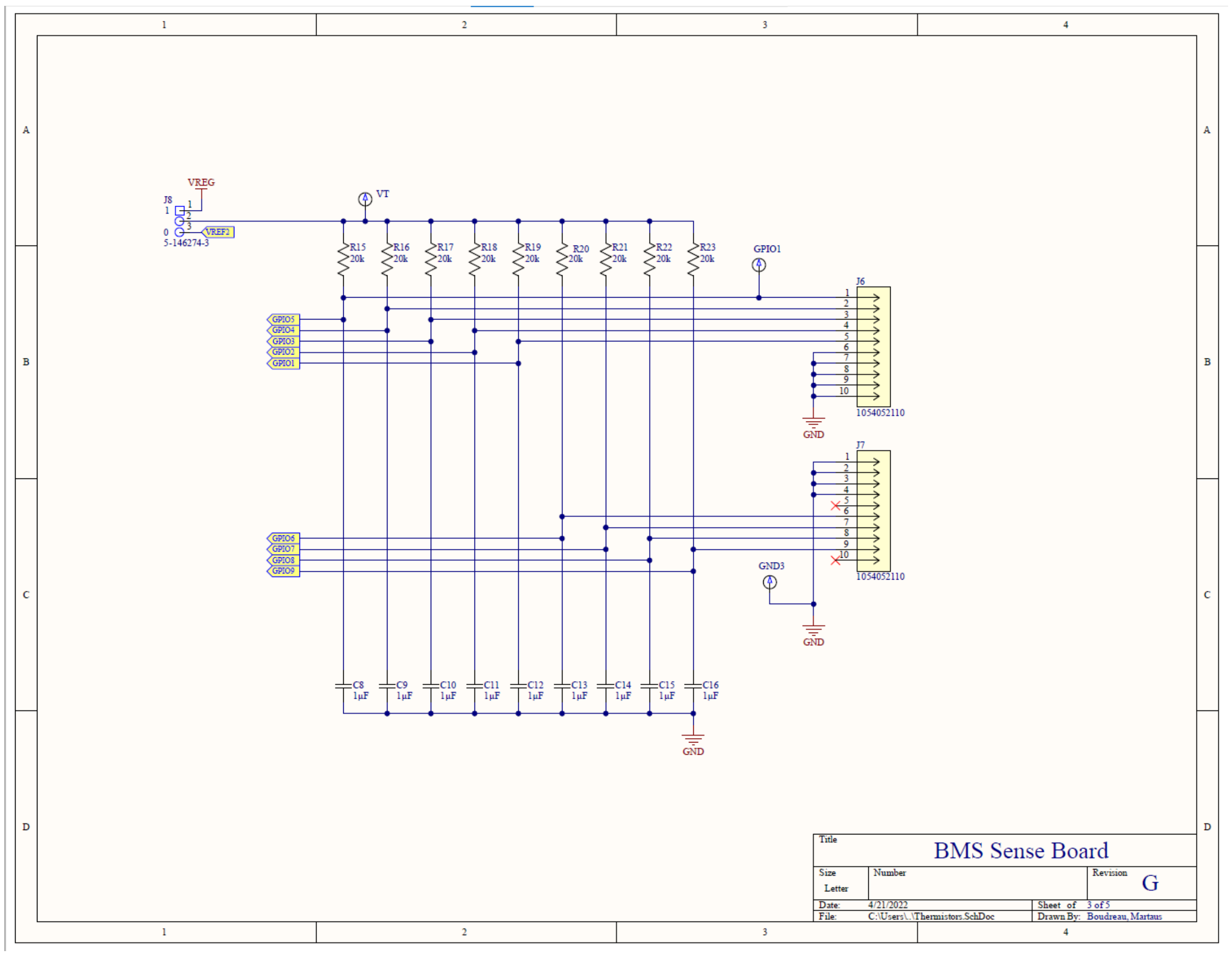


Figure 30: BMS Schematics – 3

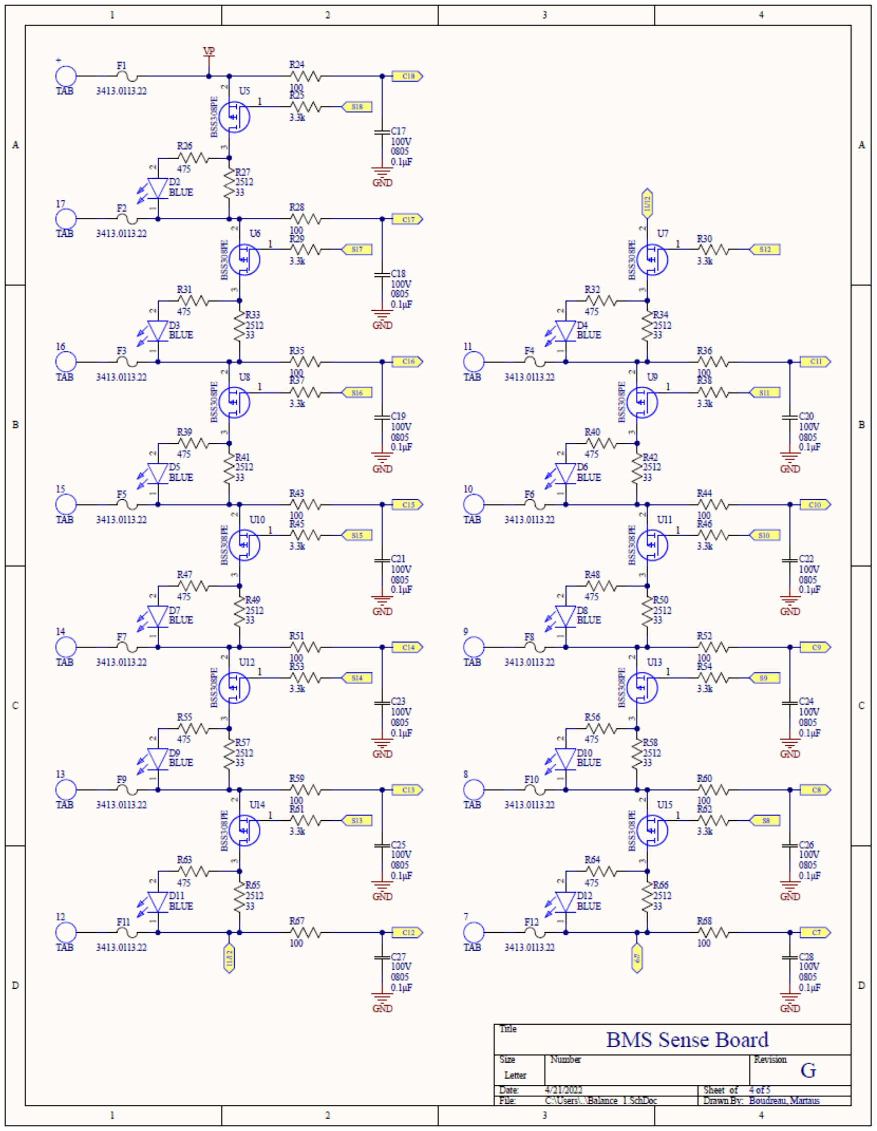


Figure 31: BMS Schematics – 4

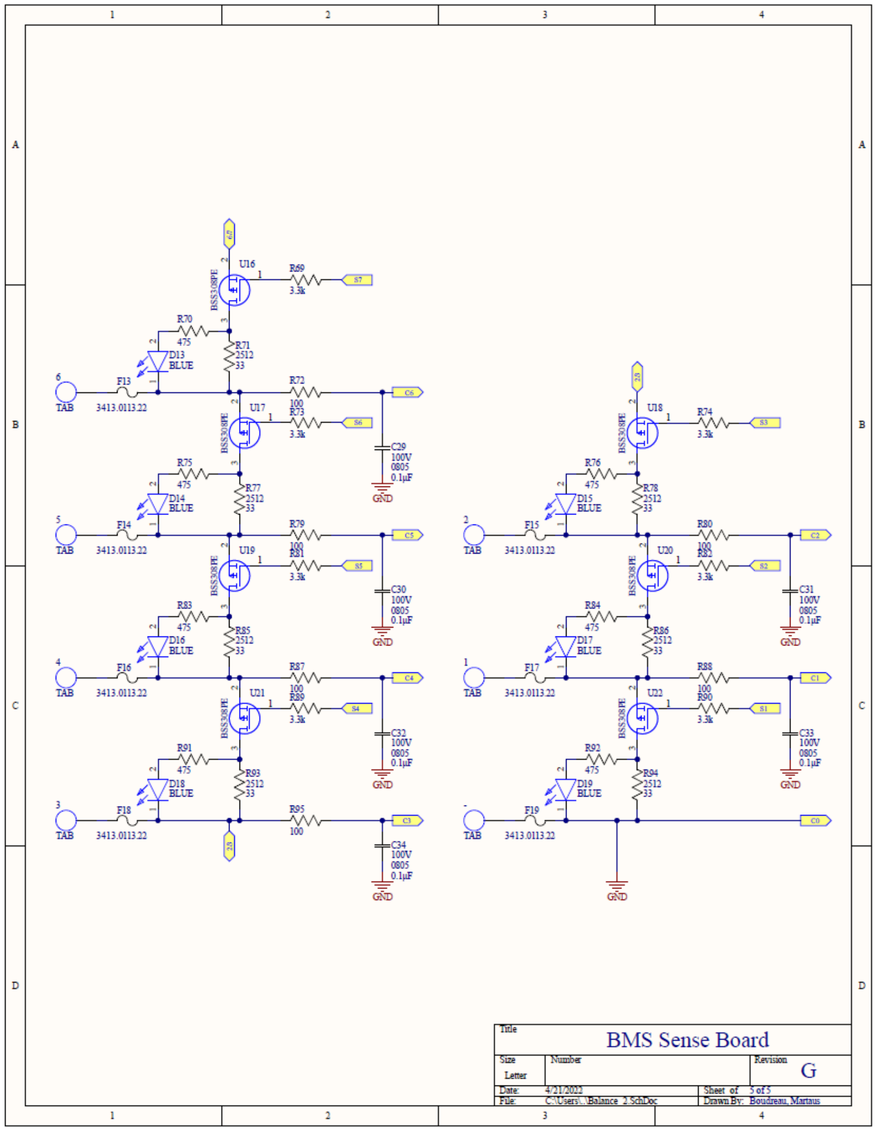


Figure 32: BMS Schematics – 5

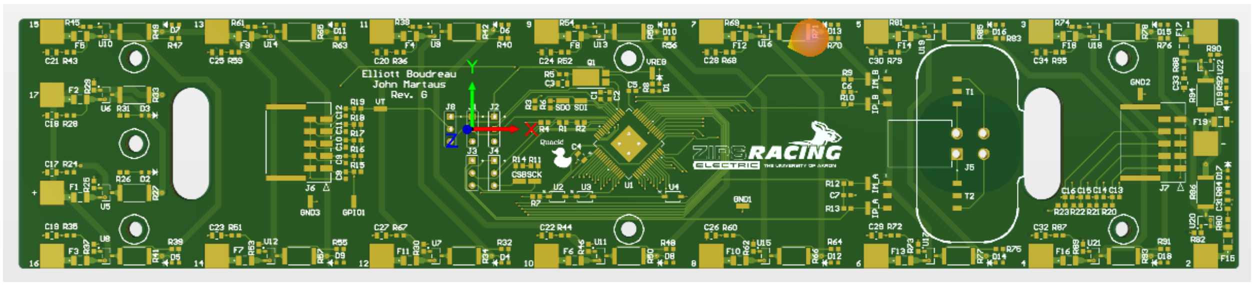


Figure 33: BMS - 3D Model

Hardware Schematics Discussion – Battery Management System (Sense and Discharge Board)

The sense and discharge board serves two main functions. The first of which is to sense the voltages and temperatures of the cells. The second purpose of the board is to discharge the cells if cell imbalance is detected. The board's architecture is built around the LTC6813HLWE-1#3ZZPBF. This is a battery management chip built by Analog Devices Incorporated. Each device monitors 18 different cells that are in series with each other. It monitors the cells voltage and relays it to the microcontroller on the AMS board through IsoSPI. The microcontroller will send a signal to charge the correct S node of the cell that needs to be discharged from there a MOSFET will be closed and current will be allowed to flow through the corresponding discharge resistor. The discharge current was calculated around a potential S.O.C. imbalance and a desired balancing time. The S.O.C. imbalance was chosen as 5% and the discharge time was selected to be 0.75 hours, arbitrarily. The balance current and balance discharge resistor are calculated as follows:

$$\frac{\%SOCImbalance * Battery\ Capacity}{Number\ of\ Hours\ to\ Balance} = \frac{5\% * 3000mAHrs}{1.5Hrs} = 100mA$$

$$\frac{Nominal\ Cell\ Voltage}{Balance\ Current} = \frac{3.6V}{100mA} = 36Ohms \approx 33Ohms$$

The 33 Ohm discharge resistors at a size 2512 were selected; a 2512 sized resistor is perfectly capable of discharging at 100mA. The LTC6813HLWE-1#3ZZPBF has internal switches that could be utilized in place of the MOSFETS that are used in the discharge loop, but the external MOSFETs were chosen to prevent the LTC6813HLWE-1#3ZZPBF from overheating. The next part of the circuit is the thermistor based temperature sensing. The thermistors are off the board making physical contact with the cells. The thermistors chosen have a temperature range of 0 to 70 degrees and a relative resistance range of 70204 to 3100.4. This resistance is used as the lower end of a voltage divider that is read by the LTC6813HLWE-1#3ZZPBF and sent to the microcontroller. Finally, the voltage is read through the sense lines seen in the schematic as the CXX pins.

The LTC6813HLWE-1#3ZZPBF requires a separate ground for every different chip used. These grounds are isolated through a 1:1 transformer that is used to also send the isoSPI signals to the microcontroller. The power provided to the LTC6813HLWE-1#3ZZPBF is from the cells it measures. It handles the input power through a DRIVE pin which acts as a discrete 5V regulator. The DRIVE pin outputs a 5.7V output capable of sourcing 1mA. This when buffered with an NPN transistor creates a steady 5 volts.

Author: EB

Battery Management System (Sense and Discharge Board)– Bill of Materials

Comment	Description	Designator	Quantity
TAB	BMS Tab	-, +, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17	19
CAP_0603_1µF	1µF Ceramic Capacitor 0603	C1, C2, C5, C8, C9, C10, C11, C12, C13, C14, C15, C16	12
CAP_0805_0.1µF	0.1µF 100V Ceramic Capacitor 0805	C3, C4, C17, C18, C19, C20, C21, C22, C23, C24, C25, C26, C27, C28, C29, C30, C31, C32, C33, C34	20
CAP_0603_10nF	10nF Ceramic Capacitor 0603	C6, C7	2
TP5015	PC TEST POINT MINIATURE	CSB, GND1, GND2, GND3, GPIO1, IM_A, IM_B, IP_A, IP_B, SCK, SDI, SDO, VREG, VT	14
GRN	Generic - Green	D1	1
BLUE	Generic - Blue	D2, D3, D4, D5, D6, D7, D8, D9, D10, D11, D12, D13, D14, D15, D16, D17, D18, D19	18
3413.0113.22	500 mA 125 V AC 125 V DC Fuse Board Mount (Cartridge Style Excluded) Surface Mount 1206 (3216 Metric)	F1, F2, F3, F4, F5, F6, F7, F8, F9, F10, F11, F12, F13, F14, F15, F16, F17, F18, F19	19
5-146274-3	AMPMODU 3 Position 2.54 mm Single Row Through-Hole Straight Breakaway Header	J1, J2, J3, J4, J8	5
1586042-4	Connector Header Through Hole, Right Angle 4 position 0.165" (4.20mm)	J5	1
1054052110	Connector Header Surface Mount, Right Angle 10 position 0.098" (2.50mm)	J6, J7	2
DXT5551P5-13	Bipolar (BJT) Transistor NPN 160 V 600 mA 130MHz 2.25 W Surface Mount PowerDI™ 5	Q1	1
RES_0603_787	RES SMD 787 OHM 1% 0603	R1	1
RES_0603_1.21k	RES SMD 1.21k OHM 1% 0603	R2	1
RES_0603_2k	RES SMD 2K OHM 1% 0603	R3, R6, R11, R14	4
RES_0603_1k	RES SMD 1K OHM 1% 0603	R4	1
RES_0603_100	RES SMD 100 OHM 1% 0603	R5, R7, R24, R28, R35, R36, R43, R44, R51, R52, R59, R60, R67, R68, R72, R79, R80, R87, R88, R95	20
RES_0603_470	RES SMD 470 OHM 1% 0603	R8	1
RES_0603_49.9	RES SMD 49.9 OHM 1% 0603	R9, R10, R12, R13	4
RES_0603_20k	RES SMD 20k OHM 1% 0603	R15, R16, R17, R18, R19, R20, R21, R22, R23	9
RES_0603_3.3k	RES SMD 3.3K OHM 1% 0603	R25, R29, R30, R37, R38, R45, R46, R53, R54, R61, R62, R69, R73, R74, R81, R82, R89, R90	18
RES_0603_475	RES SMD 475 OHM 1% 0603	R26, R31, R32, R39, R40, R47, R48, R55, R56, R63, R64, R70, R75, R76, R83, R84, R91, R92	18
RES_2512_33	RES SMD 33 OHM 1% 2512	R27, R33, R34, R41, R42, R49, R50, R57, R58, R65, R66, R71, R77, R78, R85, R86, R93, R94	18
HM2113ZNLT	150µH, 450µH - Pulse Transformer 1:1 Surface Mount	T1, T2	2
LTC6813HLWE-1#3ZZPBF	Battery Battery Monitor IC Multi-Chemistry 64-LQFP-EP (10x10)	U1	1
SZMMSZ39T1G	Zener Diode 39 V 500 mW ±5% Surface Mount SOD-123	U2, U3, U4	3
BSS308PE	P-Channel 30 V 2A (Ta) 500mW (Ta) Surface Mount PG-SOT23	U5, U6, U7, U8, U9, U10, U11, U12, U13, U14, U15, U16, U17, U18, U19, U20, U21, U22	18

Table 20: Sense Boards Bill of Materials

Software Design

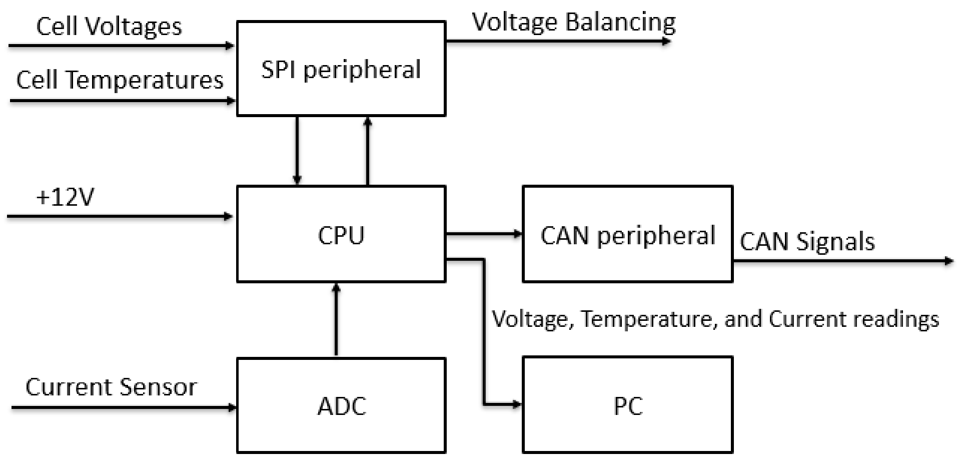


Figure 34: Software Block Diagram

Module	BMS Embedded Software
Designers	Elena Falcione
Inputs	Cell Voltages: Battery cell voltages from BMS chip array. Max 4.2V
	Cell Temperatures: Battery cell temperatures from BMS chip array
	12 Volts: Power for the CPU
	Current Sensor: Analog current data
Outputs	Voltage Balancing: Voltages of all cells are balanced. 10mV max difference
	CAN Signals: 1 MB. CAN signals sent to vehicle CAN bus
Description	The microcontroller CPU will receive cell voltages and cell temperatures from the SPI peripheral. This data will be used in an algorithm to generate SPI signals to balance cell voltages. The ADC of the microcontroller will be used to read analog current sensor data. All of the data sent to the CPU will be sent to a PC for monitoring and sent to the CAN peripheral for data logging.

Table 21: Software Functional Requirements Table

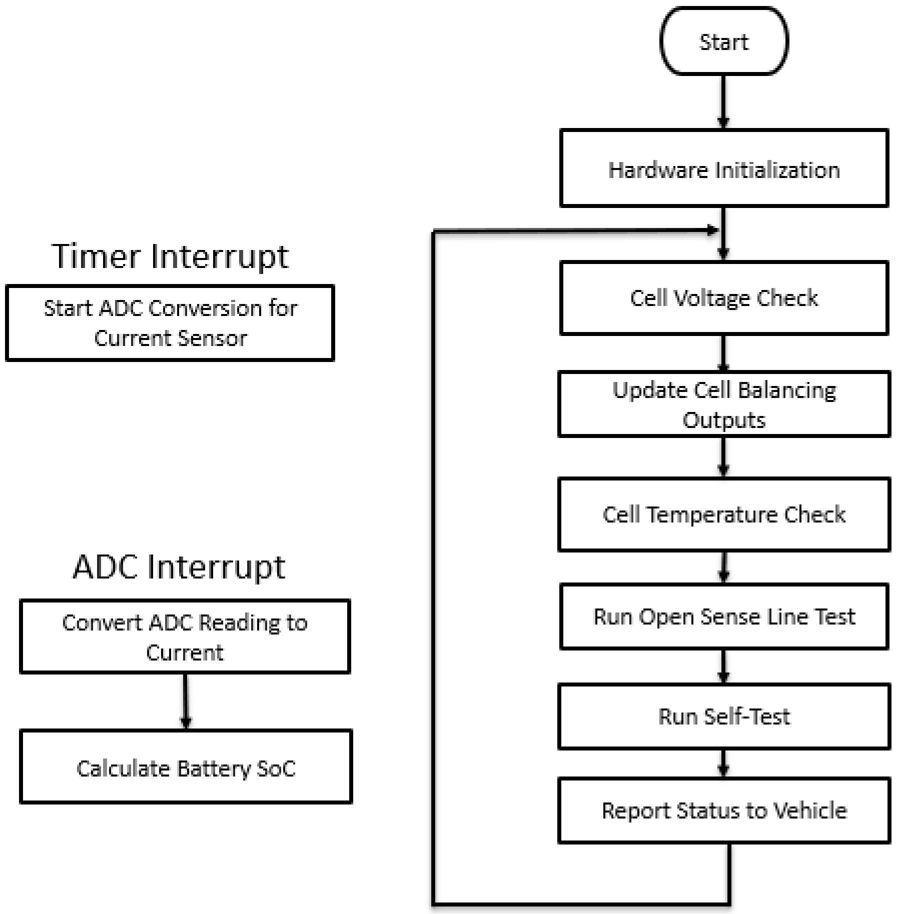


Figure 35: Embedded Software Flowchart

The microcontroller will run a main loop which requests data from the sense PCBs. If necessary, cell balancing outputs will be updated. The microcontroller also manages BMS self-maintenance, such as an open sense line test which will detect if the overcurrent protection device on a sense line has tripped. The BMS will report its status to the rest of the vehicle via CAN bus.

Author: EF

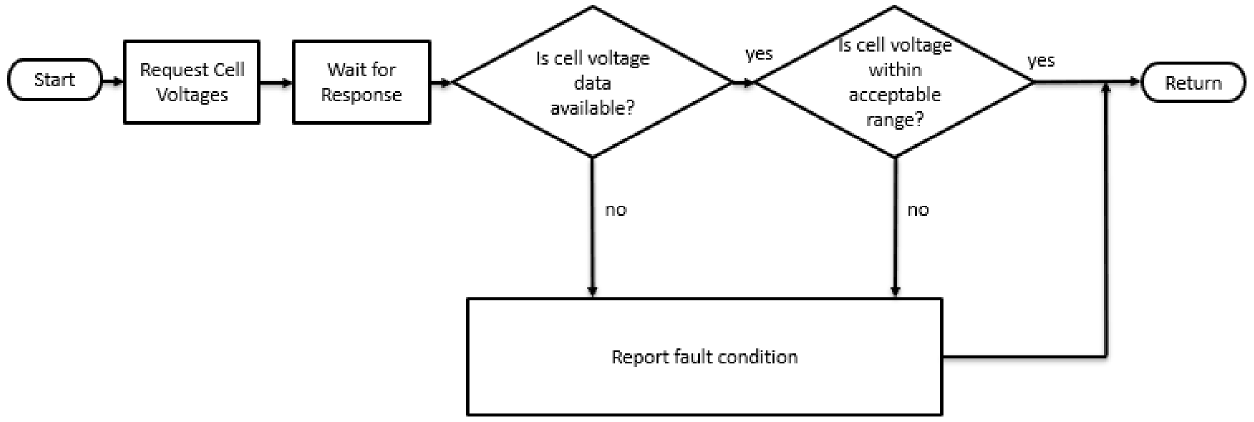


Figure 36: Embedded Software Cell Voltage Check Flowchart

A voltage measurement function is shown in Figure 31 above. The microcontroller first requests cell voltages from the sense PCBs. If a response is received, the microcontroller determines whether the cell voltages are within an acceptable range. If there is no data available or the cell voltages are outside the allowable range, a fault is reported. A cell temperature function is implemented similarly.

Author: EF

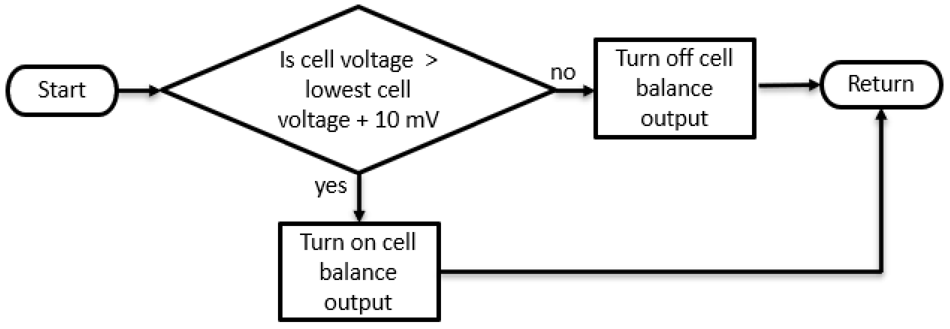

Start
Is cell voltage > lowest cell voltage + 10 mV
no
Turn off cell balance output
Return
yes
Turn on cell balance output

Figure 37: Embedded Software Cell Balance Outputs Flowchart

The cell balancing outputs are updated every time the main loop is executed. If a measured cell voltage is greater than the lowest cell voltage plus 10 mV, the balance output is turned on. When the balance output is turned on, a resistor is put in the circuit in parallel with the cell. The cell's excess energy is dissipated through the resistor. The cell continues to balance until it is within the 10 mV threshold. This ensures all the battery cells remain near the same voltage. This help ensure cell longevity, avoid excessive heat generation, and maximize the available energy.

Author: EF

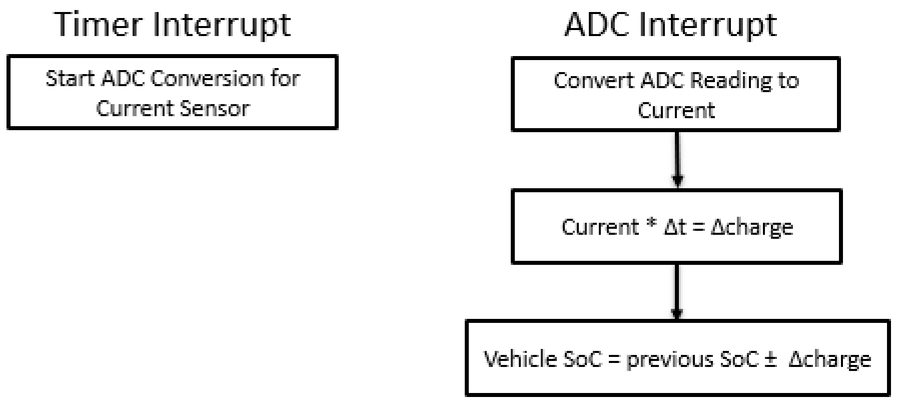

Timer Interrupt
Start ADC Conversion for Current Sensor
ADC Interrupt
Convert ADC Reading to Current
Current * Δt = Δcharge
Vehicle SoC = previous SoC ± Δcharge

Figure 38: Embedded Software Interrupt Flowchart

The Coulomb Counting SoC algorithm will be implemented as follows:

Continuous time:

$$q = \int_0^t i\,dt$$

$$q_t = c * \frac{60\ s}{1\ h}$$

$$SoC\ [\%] = SoC_0 \pm \frac{q}{q_t}$$

Discrete time:

$$q_s = i_s * \Delta t$$

$$q_t = c * \frac{60\ s}{1\ h}$$

$$SoC_n[\%] = SoC_{n-1} \pm \frac{q_s}{q_t}$$

SoC – state of charge [%]	i – current [A]	q_s – sampled charge [C]
SoC_0 – initial SoC [%]	i_s – sampled current [A]	c – total capacity [Ah]
t – time [s]	q – spent charge [C]	
Δt – sample period [s]	q_t – total capacity [C]	

In continuous time, current entering or exiting the battery pack is integrated to determine the change in stored charge. In discrete time, current entering or exiting the battery pack is sampled and multiplied by the sampling period in order to determine the change in stored charge.

A timer interrupt occurring every Δt seconds will sample the analog current sensor pin and run an ADC conversion. The stored SoC value will be updated with each sample. A flowchart describing the software architecture can be found in Figure 30.

Authors: EF

The code for the SoC calculation can be found in the appendix. A timer interrupt is triggered every 4 ms. The timer is configured with a prescaler and a period counter. The prescaler is 64 and the period counter is 624. The timer period can be calculated as follows:

$$Timer\ 1\ Frequency = \frac{F_{OSC}}{2} * \frac{1}{prescaler} = \frac{40\ [MHz]}{2} * \frac{1}{64} = 625000\ [Hz]$$

$$Timer\ 1\ Period = \frac{(period\ counter + 1)}{Timer\ 1\ Frequency} = \frac{624 +\ 1}{625000\ [Hz]} = 0.001\ [s]$$

This timer interrupt starts the ADC sampling process to measure the current sensor analog signal. When the sampling process is complete, the ADC triggers an interrupt, and the sample can be collected. To minimize the floating-point math performed in the interrupt, the samples are summed in the interrupt to be processed later. The current sensor analog output is described by the following equation:

$$Current\ Sensor\ Output\ [V] =\ 2.5\ [V] \pm 4\ [\frac{mV}{A}]$$

To convert the ADC sample to a current value in amps, several constants must be known. The ADC sample is 10 bits, so its maximum value is 1023. The analog current sensor output must be level shifted from 5 V logic down to the microcontroller 3.3 V logic level. This is achieved using a voltage divider using a 4.7 kΩ resistor and a 9.1 kΩ resistor. The ADC reference voltage is 3.3 V. The ADC samples can be converted into a current measurement using the following equation:

$$Current\ [A] = \frac{ADC\ sample\ [bits] * 3.3\ [V]}{1023\ [bits] * \frac{9.1\ k\Omega}{4.7\ k\Omega + 9.1\ k\Omega} * 4\ [\frac{mV}{A}]} - \frac{2.5\ [V]}{4\ [\frac{mV}{A}]}$$

This equation produces the current measurement which is transmitted via the CAN bus. Next, this current measurement is converted into an amp hour charge value to be used in the state of charge calculation. Let S be the summed current sensor samples and n be the number of samples included in the sum. The charge value can be calculated using the following equation:

$$Spent\ Charge\ [Ah] = \frac{3.3\ [V] * 0.001\ [ms] * 1\ [min] * 1\ [h]}{1023\ [bits] * \frac{9.1\ k\Omega}{4.7\ k\Omega + 9.1\ k\Omega} * 4\ \left[\frac{mV}{A}\right] * 60\ [s] * 60\ [min]} * S - \frac{0.001\ [ms] * 1\ [min] * 1\ [h] * 2.5\ [V]}{4\ \left[\frac{mV}{A}\right] * 60\ [s] * 60\ [min]} * n$$

Author: EF

Battery pack state of charge must be maintained over system power cycles. An external EEPROM chip is used to store SoC. When the BMS is powered on or reset, four bytes are read from the EEPROM. The first two bytes are an “is valid” code, which indicates whether the second two bytes contain valid SoC data. If the memory became corrupted, these two bytes would allow the microcontroller to recognize that the stored SoC value is invalid. The second two bytes indicate state of charge times ten. The read value is used as the initial state of charge, which is adjusted as necessary by the ADC sampling interrupt. During BMS operation, the SoC is periodically updated on the EEPROM chip. This way, the EEPROM chip contains the most up to date information in case of unexpected power loss.

Author: EF

To perform the open sense line test and the self-test, a message must be sent to each LTC6813 chip via isoSPI. The last step in the main while loop is to output a status message to the main vehicle CAN bus. Pseudocode for this process can be found below:

```
1) transmit ADOW open sense line command
2) wait for response
3) if response received
    a. parse received data
    b. if any sense line has gone open
        i. add cell ID number to status message
4) else
    a. add fault code to status message
5) transmit CVST, AXST, and DIAGN self-test commands
6) wait for response
7) if response received
    a. parse received data
    b. if any self-test fails
        i. add fault code to status message
8) else
    a. add fault code to status message
9) output status CAN message to main CAN bus
```

Author: EF

The CVST and AXST commands perform ADC self-tests. The LTC6813 chip produces test signals which are fed into the digital filter circuitry. The CVST command tests the ADCs used for cell voltage measurement, and the self-test output is stored in the cell voltage registers. The AXST command tests the ADCs used for GPIO measurements, and the self-test output is stored in the auxiliary pin registers. The results of both self-tests can be read and compared to lookup table value. If the read value matches the expected value from the lookup table, the self-test was successful.

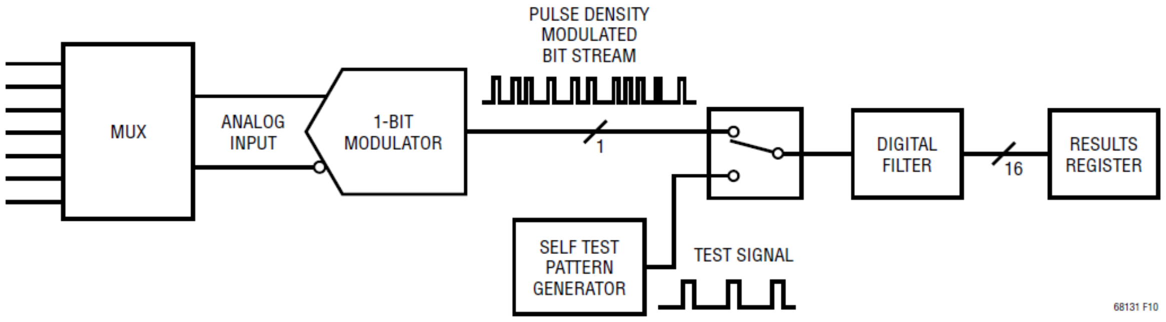


Figure 39: ADC Self Test

Author: EF

The DIAGN command is used to perform a multiplexer self-test. When the LTC6813 chip receives the DIAGN command, it cycles through each of its multiplexer channels to ensure the corresponding data is passed through the multiplexer. If any of the operations fail, a single bit is set in Status Register B to indicate the failure. The CVST, AXST, and DIAGN commands satisfy the engineering requirement that the BMS must be able to detect internal faults.

Author: EF

Every SPI command used by the BMS is post-fixed with a 2-byte packet error code (PEC). The PEC is a 15-bit cyclic redundancy check (CRC) which is calculated based on the data being transmitted. The BMS microcontroller must calculate PECs to be transmitted with sent commands. The BMS microcontroller also calculates PECs based on received data and compares the calculated version to the received PEC. If the calculated and received PECs differ, the BMS microcontroller concludes that some bit corruption has occurred and throws out the data. From there, retransmission can be requested. Shielded cable is used to minimize the likelihood of this occurring, although it is possible.

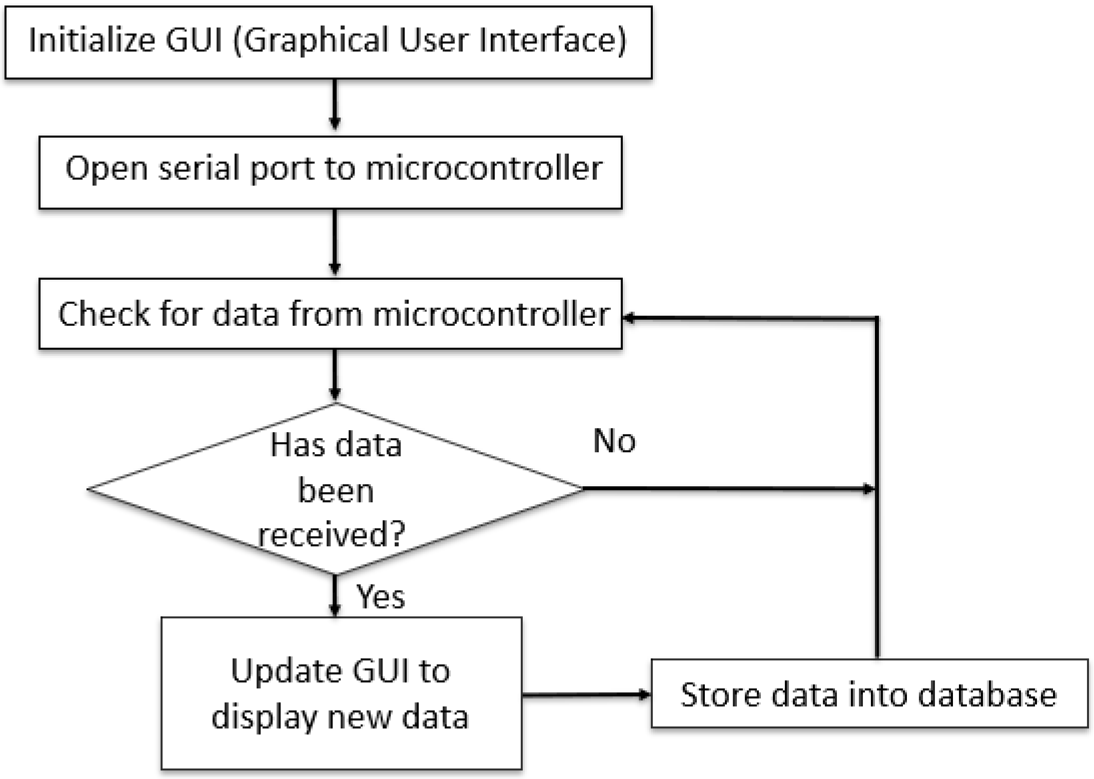

Initialize GUI (Graphical User Interface)
Open serial port to microcontroller
Check for data from microcontroller
Has data been received?
No
Yes
Update GUI to display new data
Store data into database

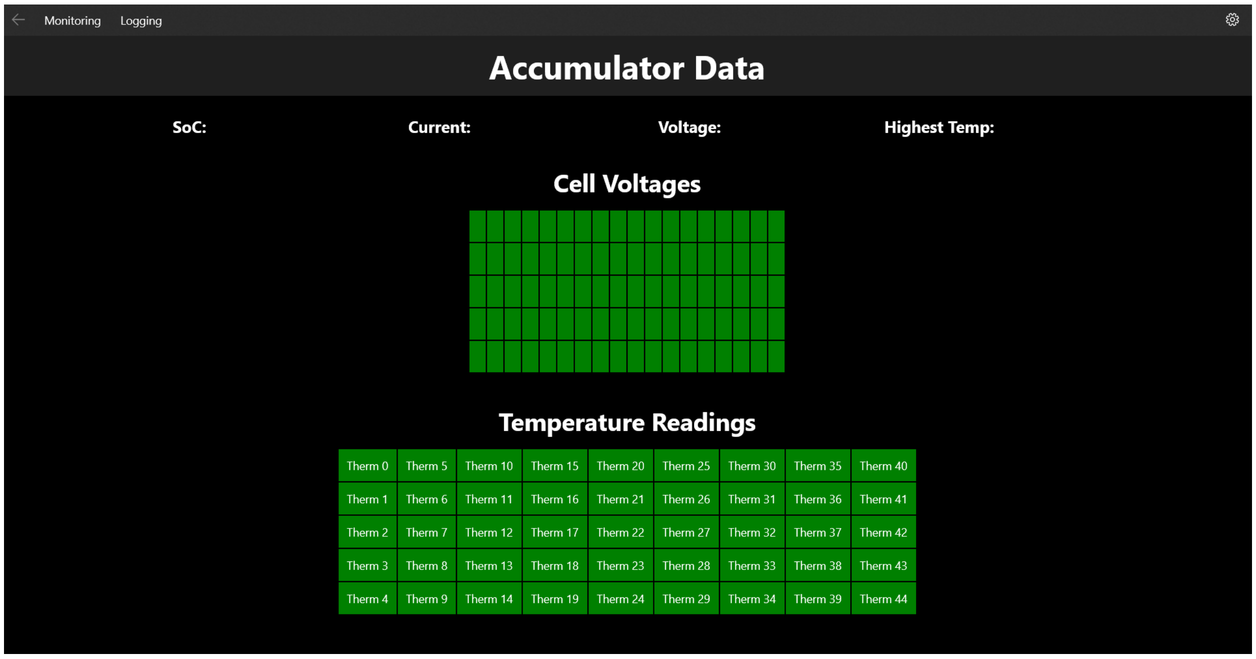
Figure 40: BMS Monitoring Application Software Flowchart

The BMS Monitoring System (BMSMS) application (as shown in Figure 34) will receive temperature, voltage, current readings, and state of charge from the BMS CAN bus and display the data in a graphical user interface (see Figure 35) on a PC. Code for the BMSMS can be found in the appendix.

The BMSMS was designed using the model view viewmodel design pattern (described in Design Methods**:**). The program runs with two main threads: the graphical user interface (GUI) thread and a CAN bus listener thread. The GUI thread manages all updates and interactions with the GUI. The CAN bus listener thread runs in the background, monitoring the connected CAN bus for temperature, voltage, and current sensor reading changes. The CAN bus is accessed by connecting a Kvaser Leaf Semipro CAN-to-USB device to the PC and the CAN bus, which is then interfaced with the BMSMS through the Kvaser Canlib C# library. The CAN listener thread will automatically connect to the Kvaser tool when it is plugged in and will detect if the device is in an error state or unplugged then reconnect to the device if the issue is resolved. This is done by polling the USB interfaces on the PC every five seconds until a valid Kvaser tool is detected and connected to. If an error is detected with the device or the device is unplugged, the thread will disconnect the device and attempt to connect again every five seconds until a valid device is found.

When a message is read from the CAN bus, the CAN listener thread will parse the message to determine what information it contains (cell voltage, thermistor temperature, current sensor reading, etc.) then it will store the received data in the database and send a request to the GUI thread to update the relevant data displayed in the user interface. The data that is collected by the CAN listener thread and stored in the database can be viewed and queried in the 'Logging' page of the BMSMS.

Author: DD



Accumulator Data

SoC: Current: Voltage: Highest Temp:

Cell Voltages

Temperature Readings

Therm 0	Therm 5	Therm 10	Therm 15	Therm 20	Therm 25	Therm 30	Therm 35	Therm 40
Therm 1	Therm 6	Therm 11	Therm 16	Therm 21	Therm 26	Therm 31	Therm 36	Therm 41
Therm 2	Therm 7	Therm 12	Therm 17	Therm 22	Therm 27	Therm 32	Therm 37	Therm 42
Therm 3	Therm 8	Therm 13	Therm 18	Therm 23	Therm 28	Therm 33	Therm 38	Therm 43
Therm 4	Therm 9	Therm 14	Therm 19	Therm 24	Therm 29	Therm 34	Therm 39	Therm 44

Figure 41: BMSMS GUI

Mechanical Sketch

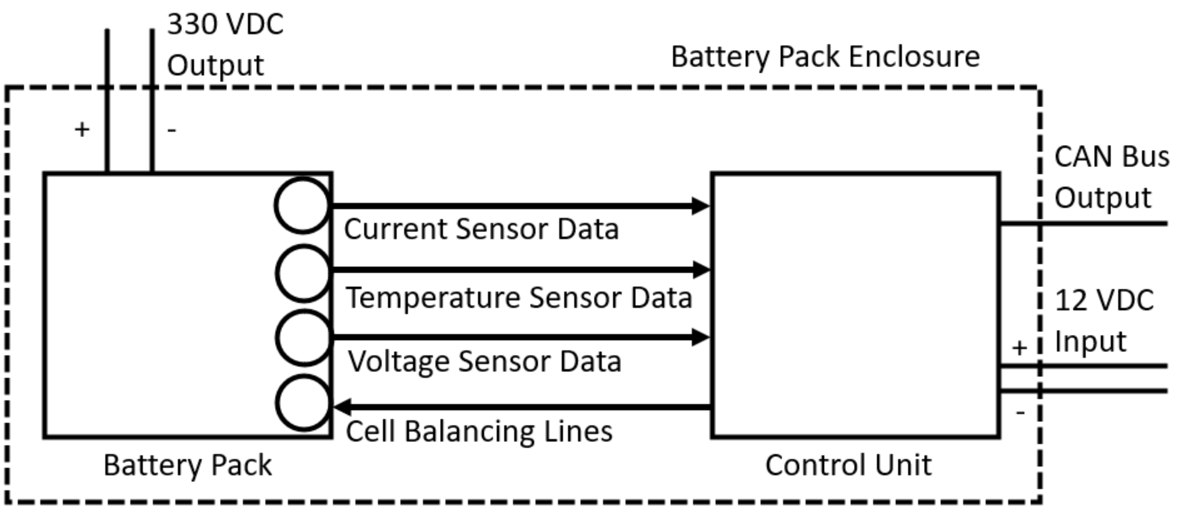


Figure 42: Mechanical Sketch

Author: EF

As shown in Figure 36, the Battery pack and the BMS control unit will reside within the battery pack enclosure.

Author: DD

Budget Information

Item	Qty	Unit Cost	Total Cost
DM330026	1	$ 49.97	$ 49.97
B160B-13-F	30	$ 0.74	$ 22.20
1SMA4758-GT3TR	10	$ 0.42	$ 4.20
V23105A5003A201	15	$ 3.76	$ 56.40
TMR 1-1211	5	$ 9.20	$ 46.00
LR8N3-G	10	$ 0.72	$ 7.20
UWX1E560MCL1GB	10	$ 0.41	$ 4.10
TNPW0805249KBEEA	20	$ 0.78	$ 15.60
TNPW0805100KBEEN	20	$ 0.92	$ 18.40
TNPW080510K0BEEN	10	$ 0.92	$ 9.20
EL3H7(TA)-VG	10	$ 0.49	$ 4.90
AT25040B-XHL-T	10	$ 0.32	$ 3.20
HVCB1206KDM470K	10	$ 2.67	$ 26.70
HVCB0805FDD499K	10	$ 5.83	$ 58.30
0366380002	2	$ 11.73	$ 23.46
LTST-C190TBKT	25	$ 0.35	$ 8.75
LTST-C190KFKT	30	$ 0.23	$ 6.90
LMC7221BIM5	25	$ 2.39	$ 59.75
MRJR336001	5	$ 13.09	$ 65.45
		Total:	$ 490.68

Table 22: Budget Information

Additional materials, such as development boards for the LTC6813 chips, were donated to Zips Racing Electric by various vendors. Thank you to all our sponsors, especially Analog Devices.

Team Information

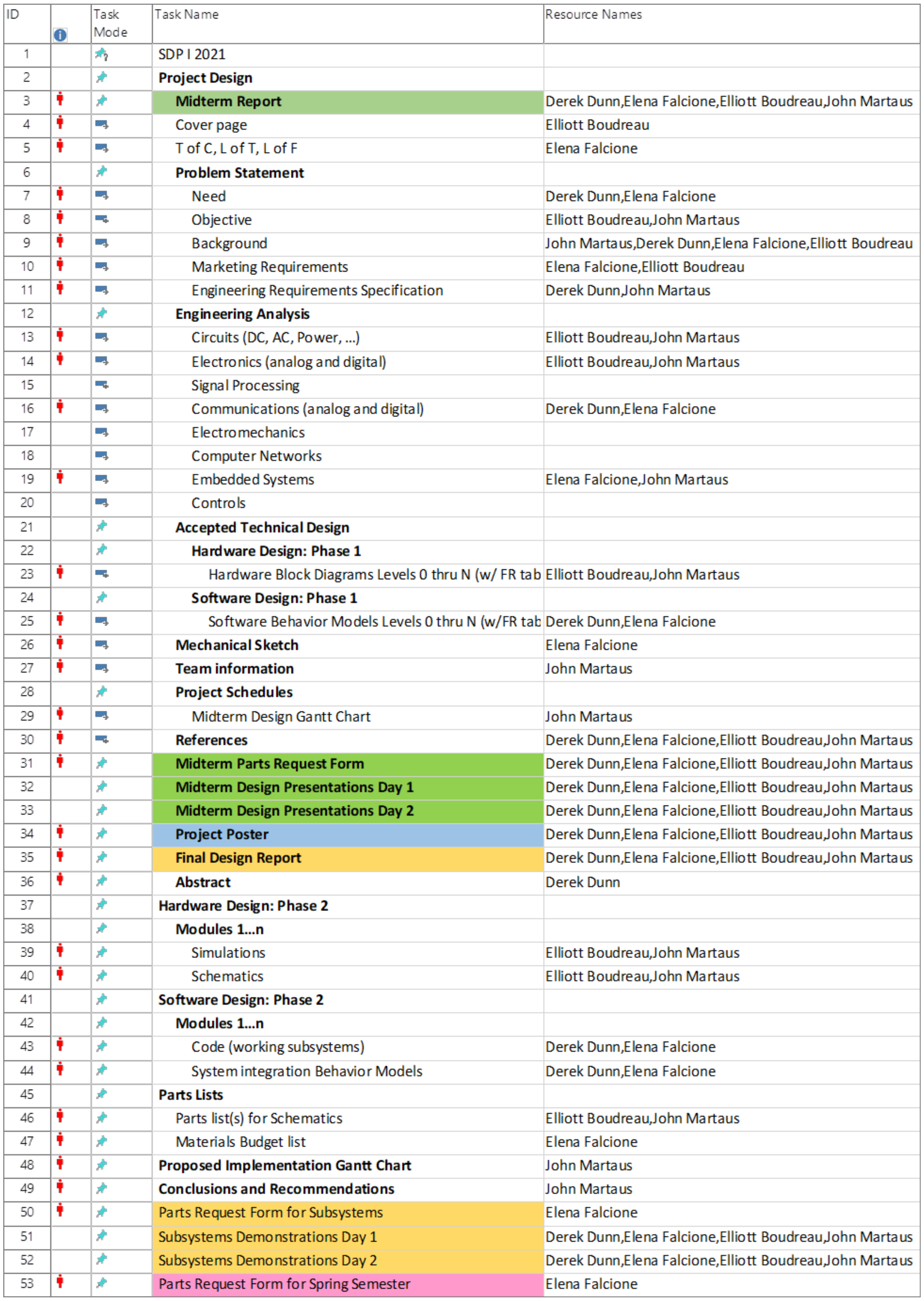
Elliott Boudreau - EE, Documentation and cell-balancing lead

Derek Dunn - CpE, Project manager and user interface lead

Elena Falcione - CpE, Software and embedded systems lead

John Martaus - EE, Hardware and sensing lead

Project Schedules



ID	●	Task Mode	Task Name	Resource Names
1		📌?	SDP I 2021	
2		📌	**Project Design**	
3	●	📌	**Midterm Report**	Derek Dunn,Elena Falcione,Elliott Boudreau,John Martaus
4	●	➡	Cover page	Elliott Boudreau
5	●	➡	T of C, L of T, L of F	Elena Falcione
6		📌	**Problem Statement**	
7	●	➡	Need	Derek Dunn,Elena Falcione
8	●	➡	Objective	Elliott Boudreau,John Martaus
9	●	➡	Background	John Martaus,Derek Dunn,Elena Falcione,Elliott Boudreau
10	●	➡	Marketing Requirements	Elena Falcione,Elliott Boudreau
11	●	➡	Engineering Requirements Specification	Derek Dunn,John Martaus
12		📌	**Engineering Analysis**	
13	●	➡	Circuits (DC, AC, Power, ...)	Elliott Boudreau,John Martaus
14	●	➡	Electronics (analog and digital)	Elliott Boudreau,John Martaus
15		➡	Signal Processing	
16	●	➡	Communications (analog and digital)	Derek Dunn,Elena Falcione
17		➡	Electromechanics	
18		➡	Computer Networks	
19	●	➡	Embedded Systems	Elena Falcione,John Martaus
20		➡	Controls	
21		📌	**Accepted Technical Design**	
22		📌	**Hardware Design: Phase 1**	
23	●	➡	Hardware Block Diagrams Levels 0 thru N (w/ FR tab	Elliott Boudreau,John Martaus
24		📌	**Software Design: Phase 1**	
25	●	➡	Software Behavior Models Levels 0 thru N (w/FR tab	Derek Dunn,Elena Falcione
26	●	➡	**Mechanical Sketch**	Elena Falcione
27	●	➡	**Team information**	John Martaus
28		📌	**Project Schedules**	
29	●	➡	Midterm Design Gantt Chart	John Martaus
30	●	➡	**References**	Derek Dunn,Elena Falcione,Elliott Boudreau,John Martaus
31	●	📌	**Midterm Parts Request Form**	Derek Dunn,Elena Falcione,Elliott Boudreau,John Martaus
32		📌	**Midterm Design Presentations Day 1**	Derek Dunn,Elena Falcione,Elliott Boudreau,John Martaus
33		📌	**Midterm Design Presentations Day 2**	Derek Dunn,Elena Falcione,Elliott Boudreau,John Martaus
34	●	📌	**Project Poster**	Derek Dunn,Elena Falcione,Elliott Boudreau,John Martaus
35	●	📌	**Final Design Report**	Derek Dunn,Elena Falcione,Elliott Boudreau,John Martaus
36	●	📌	**Abstract**	Derek Dunn
37		📌	**Hardware Design: Phase 2**	
38		📌	**Modules 1...n**	
39	●	📌	Simulations	Elliott Boudreau,John Martaus
40	●	📌	Schematics	Elliott Boudreau,John Martaus
41		📌	**Software Design: Phase 2**	
42		📌	**Modules 1...n**	
43	●	📌	Code (working subsystems)	Derek Dunn,Elena Falcione
44	●	📌	System integration Behavior Models	Derek Dunn,Elena Falcione
45		📌	**Parts Lists**	
46	●	📌	Parts list(s) for Schematics	Elliott Boudreau,John Martaus
47	●	📌	Materials Budget list	Elena Falcione
48	●	📌	**Proposed Implementation Gantt Chart**	John Martaus
49	●	📌	**Conclusions and Recommendations**	John Martaus
50	●	📌	Parts Request Form for Subsystems	Elena Falcione
51		📌	Subsystems Demonstrations Day 1	Derek Dunn,Elena Falcione,Elliott Boudreau,John Martaus
52		📌	Subsystems Demonstrations Day 2	Derek Dunn,Elena Falcione,Elliott Boudreau,John Martaus
53	●	📌	Parts Request Form for Spring Semester	Elena Falcione

Conclusions and Recommendations

The battery management system hardware and software requirements have been finalized. The team has shown the requirements and the systems that must be implemented through block diagrams and flowcharts. Components that must meet the functional requirements have been chosen and implemented on a circuit schematic. Engineering analysis on the circuit has been done to ensure the correct operation of the battery management circuitry, including cell state of charge calculations, cell balancing, and temperature sensing.

PCB layouts have been generated from the schematics and code has been developed to satisfy all engineering requirements. The next project phase is integration. The BMS must be installed in the racecar to ensure the safety of the driver and the team.

Author: JM, EF

References

AST-CAN485 Hookup Guide, Sparkfun, https://learn.sparkfun.com/tutorials/ast-can485-hookup-guide/introduction-to-can-bus.

Ivanov, Andrei Vladimirovich. Accumulator Battery Management System. Patent WO2014209161A1. 31 Dec. 2014. Print.

Loncarevic, Ivan. Battery Management System with Temperature Sensing and Charge Management for Individual Series-connected Battery Cells. Patent US10862317B2. 08 Dec. 2020. Print.

Lu, Daban, et al. "Thermal Behavior and Failure Mechanism of Large Format Lithium-Ion Battery." Journal of Solid State Electrochemistry: Current Research and Development in Science and Technology, vol. 25, no. 1, 2021, p. 315. EBSCOhost, doi:10.1007/s10008-020-04810-z.

N. Tarle, R. Kulkarni, N. Desale and V. Pawar, "Design of a Battery Management System for Formula Student Electric Race Vehicle," 2019 5th International Conference On Computing, Communication, Control And Automation (ICCUBEA), Pune, India, 2019, pp. 1-6, doi: 10.1109/ICCUBEA47591.2019.9128526.

Omariba, Zachary Bosire, et al. "Review of Battery Cell Balancing Methodologies for Optimizing Battery Pack Performance in Electric Vehicles." *IEEE Access*, vol. 7, 2019, pp. 129335–129352., doi:10.1109/access.2019.2940090.

Pop, Valer, et al. "State-of-the-Art of Battery State-of-Charge Determination." Battery Management Systems: Accurate State-of-charge Indication for Battery Powered Applications. Dordrecht: Springer, 2010. 11-16. Print.

S, Hemavathi. "Overview of Cell Balancing Methods FOR Li-Ion Battery Technology."

Energy Storage, 2020, doi:10.1002/est2.203.

Welsh, James. "A Comparison of Active and Passive Cell Balancing Techniques for Series/Parallel Battery Packs." Electronic Thesis or Dissertation. Ohio State University, 2009. *OhioLINK Electronic Theses and Dissertations Center*. 22 Mar 2021

Appendices

Glossary:

Accumulator – High voltage, lithium-ion battery pack that provides tractive power to the vehicle

AMS – Accumulator Management System

ADC – Analog to Digital Converter

BMS – Battery Management System

BMSMS – Battery Management System Monitoring System

CAN – Control Area Network

CRC – Cyclic Redundancy Check

ECU – Electronic Control Unit

GUI – Graphical User Interface

MVVM – Model View Viewmodel

SAE – Society of Automotive Engineers

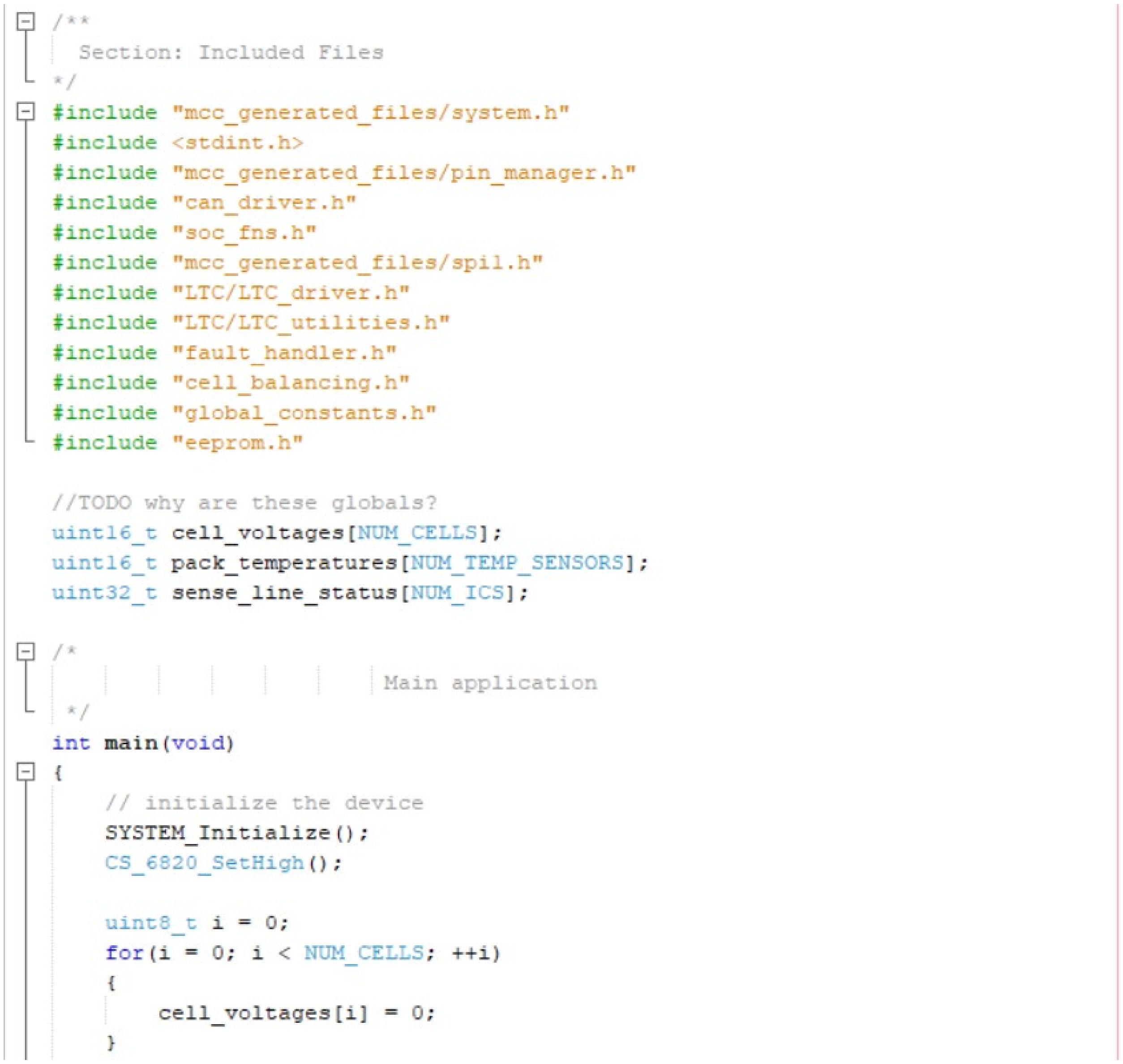
SOC – State of Charge

UI – User Interface

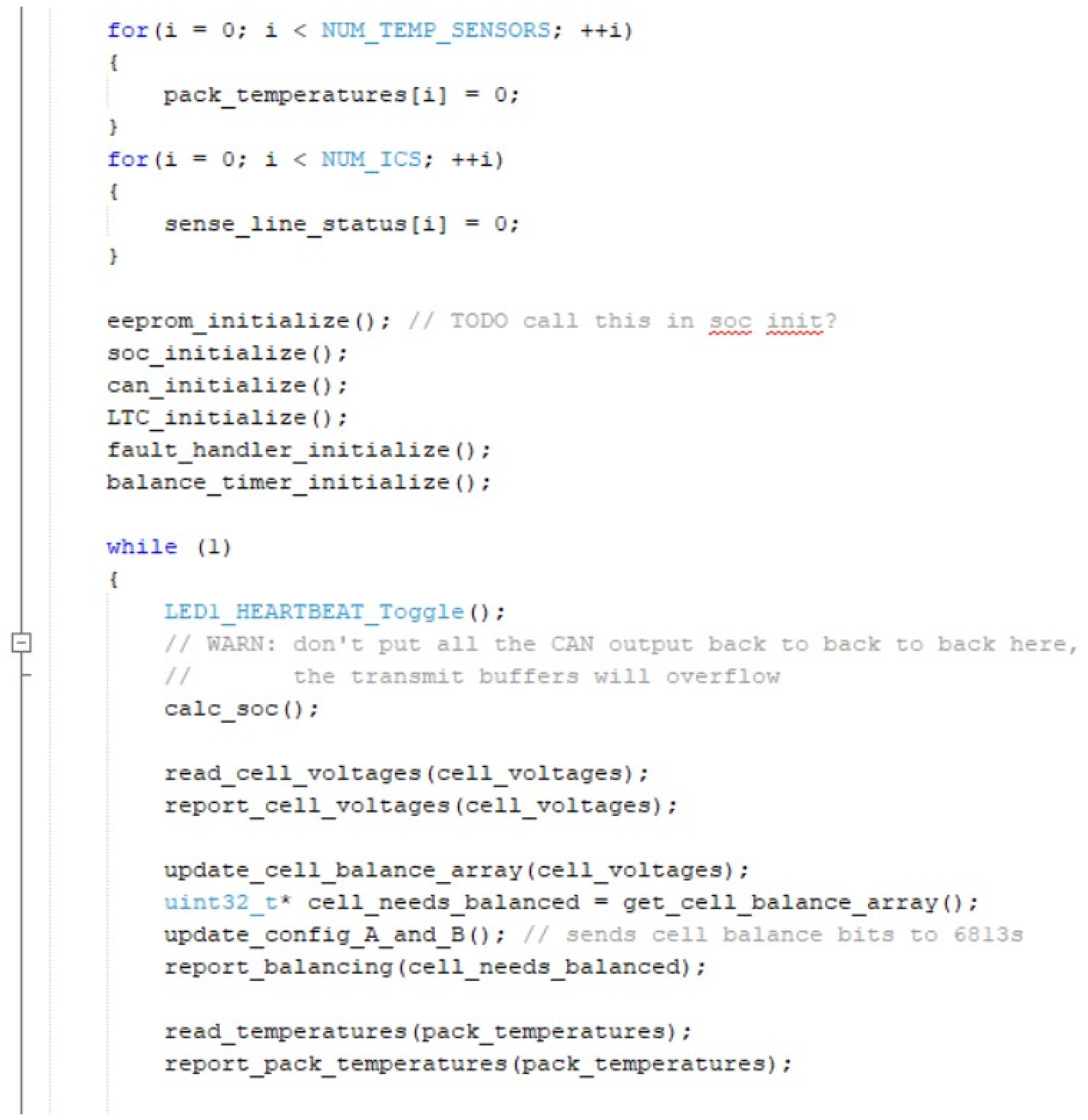
ZRE – Zips Racing Electric

Embedded Systems Code:

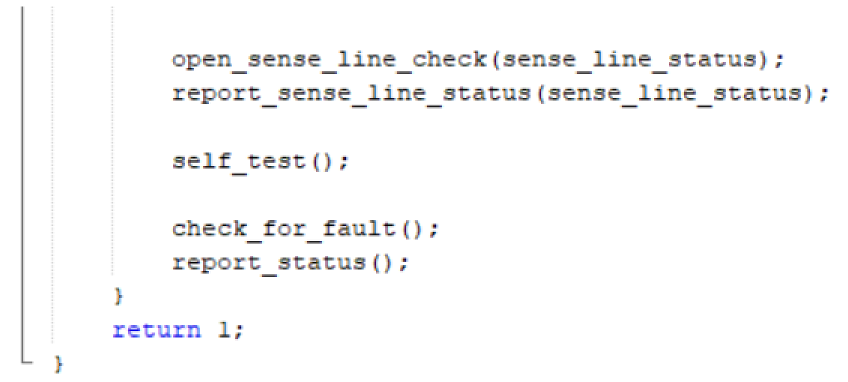
Main while loop:



```
/**
  Section: Included Files
*/
#include "mcc_generated_files/system.h"
#include <stdint.h>
#include "mcc_generated_files/pin_manager.h"
#include "can_driver.h"
#include "soc_fns.h"
#include "mcc_generated_files/spi1.h"
#include "LTC/LTC_driver.h"
#include "LTC/LTC_utilities.h"
#include "fault_handler.h"
#include "cell_balancing.h"
#include "global_constants.h"
#include "eeprom.h"

//TODO why are these globals?
uint16_t cell_voltages[NUM_CELLS];
uint16_t pack_temperatures[NUM_TEMP_SENSORS];
uint32_t sense_line_status[NUM_ICS];

/*
                         Main application
 */
int main(void)
{
    // initialize the device
    SYSTEM_Initialize();
    CS_6820_SetHigh();

    uint8_t i = 0;
    for(i = 0; i < NUM_CELLS; ++i)
    {
        cell_voltages[i] = 0;
    }
```

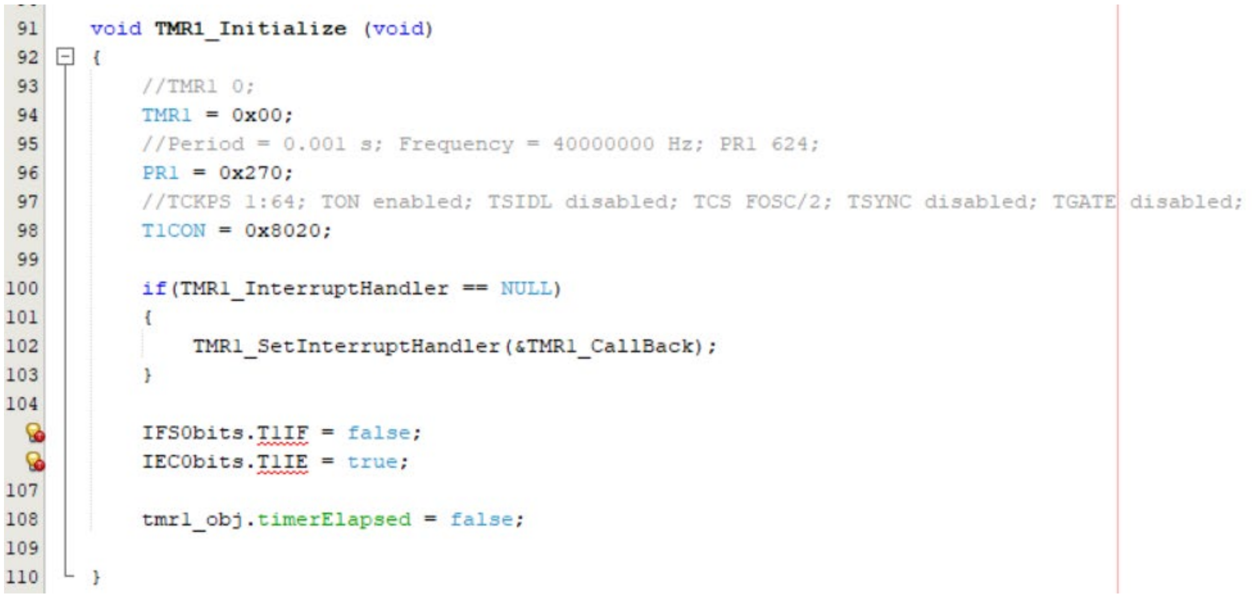


```c
    for(i = 0; i < NUM_TEMP_SENSORS; ++i)
    {
        pack_temperatures[i] = 0;
    }
    for(i = 0; i < NUM_ICS; ++i)
    {
        sense_line_status[i] = 0;
    }

    eeprom_initialize(); // TODO call this in soc init?
    soc_initialize();
    can_initialize();
    LTC_initialize();
    fault_handler_initialize();
    balance_timer_initialize();

    while (1)
    {
        LED1_HEARTBEAT_Toggle();
        // WARN: don't put all the CAN output back to back to back here,
        //       the transmit buffers will overflow
        calc_soc();

        read_cell_voltages(cell_voltages);
        report_cell_voltages(cell_voltages);

        update_cell_balance_array(cell_voltages);
        uint32_t* cell_needs_balanced = get_cell_balance_array();
        update_config_A_and_B(); // sends cell balance bits to 6813s
        report_balancing(cell_needs_balanced);

        read_temperatures(pack_temperatures);
        report_pack_temperatures(pack_temperatures);

```



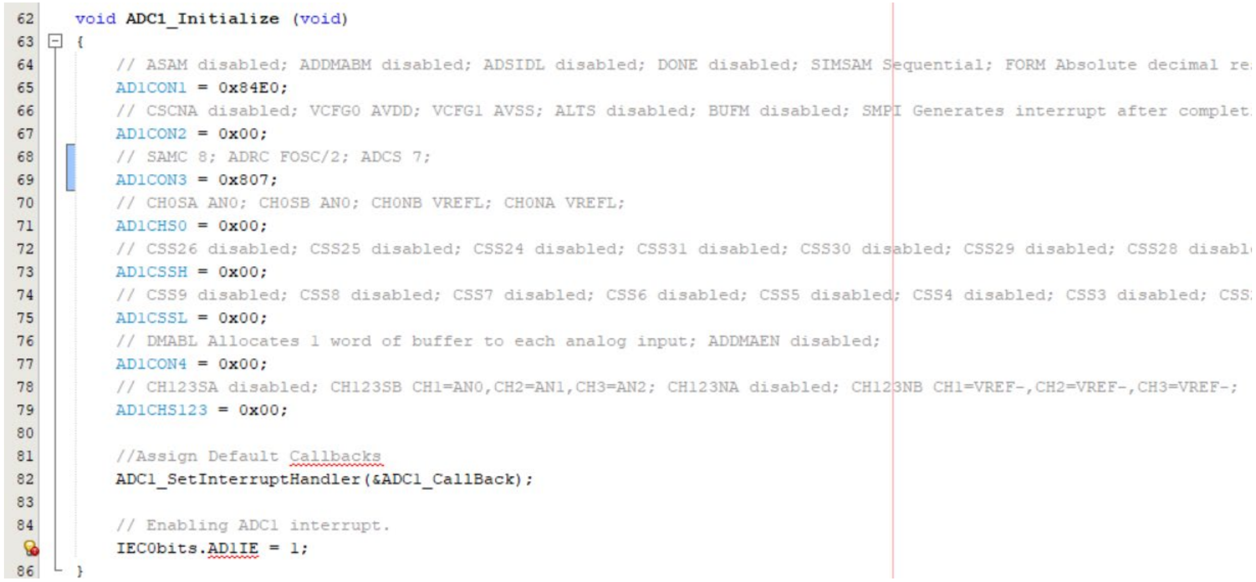
```c

        open_sense_line_check(sense_line_status);
        report_sense_line_status(sense_line_status);

        self_test();

        check_for_fault();
        report_status();
    }
    return 1;
}
```

Timer 1 initialization:



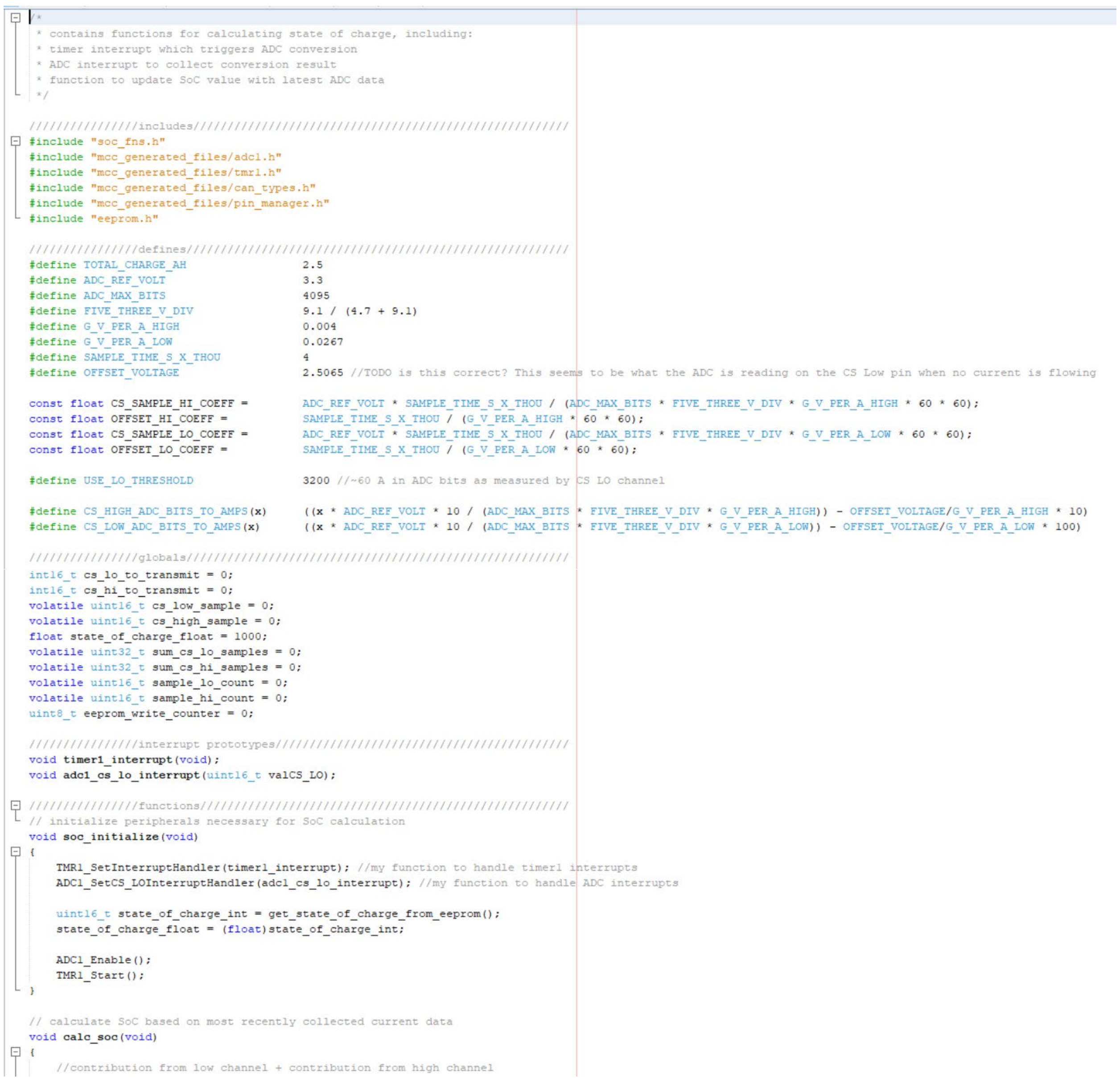
```c
void TMR1_Initialize (void)
{
    //TMR1 0;
    TMR1 = 0x00;
    //Period = 0.001 s; Frequency = 40000000 Hz; PR1 624;
    PR1 = 0x270;
    //TCKPS 1:64; TON enabled; TSIDL disabled; TCS FOSC/2; TSYNC disabled; TGATE disabled;
    T1CON = 0x8020;

    if(TMR1_InterruptHandler == NULL)
    {
        TMR1_SetInterruptHandler(&TMR1_CallBack);
    }

    IFS0bits.T1IF = false;
    IEC0bits.T1IE = true;

    tmr1_obj.timerElapsed = false;

}
```

ADC initialization:

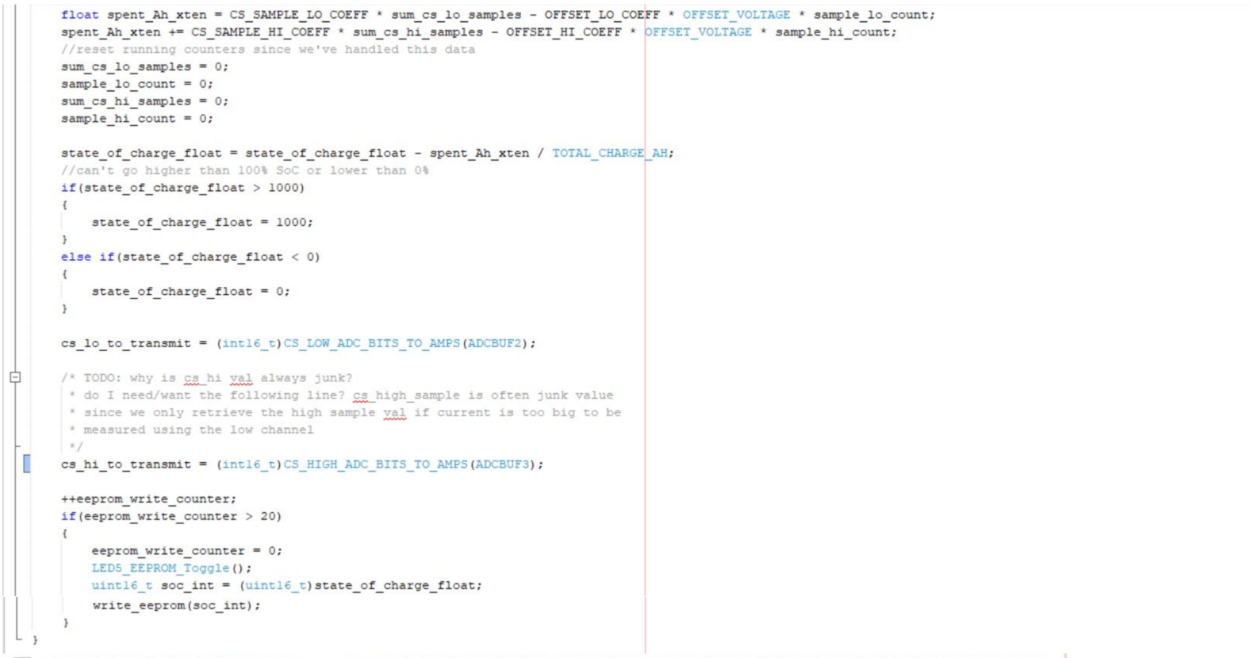


```c
void ADC1_Initialize (void)
{
    // ASAM disabled; ADDMABM disabled; ADSIDL disabled; DONE disabled; SIMSAM Sequential; FORM Absolute decimal re
    AD1CON1 = 0x84E0;
    // CSCNA disabled; VCFG0 AVDD; VCFG1 AVSS; ALTS disabled; BUFM disabled; SMPI Generates interrupt after complet
    AD1CON2 = 0x00;
    // SAMC 8; ADRC FOSC/2; ADCS 7;
    AD1CON3 = 0x807;
    // CH0SA AN0; CH0SB AN0; CH0NB VREFL; CH0NA VREFL;
    AD1CHS0 = 0x00;
    // CSS26 disabled; CSS25 disabled; CSS24 disabled; CSS31 disabled; CSS30 disabled; CSS29 disabled; CSS28 disabl
    AD1CSSH = 0x00;
    // CSS9 disabled; CSS8 disabled; CSS7 disabled; CSS6 disabled; CSS5 disabled; CSS4 disabled; CSS3 disabled; CSS
    AD1CSSL = 0x00;
    // DMABL Allocates 1 word of buffer to each analog input; ADDMAEN disabled;
    AD1CON4 = 0x00;
    // CH123SA disabled; CH123SB CH1=AN0,CH2=AN1,CH3=AN2; CH123NA disabled; CH123NB CH1=VREF-,CH2=VREF-,CH3=VREF-;
    AD1CHS123 = 0x00;

    //Assign Default Callbacks
    ADC1_SetInterruptHandler(&ADC1_CallBack);

    // Enabling ADC1 interrupt.
    IEC0bits.AD1IE = 1;
}
```

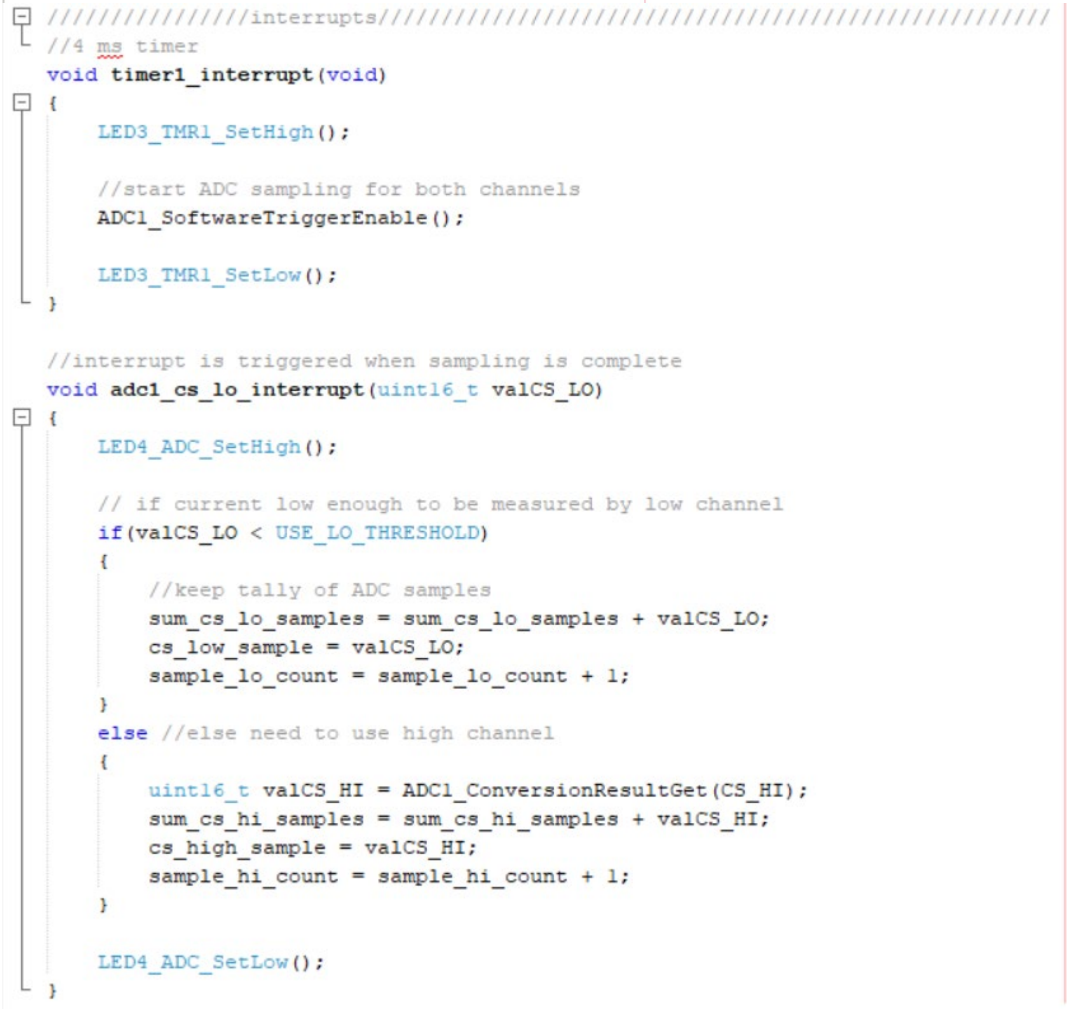
SoC Algorithm:



```
/*
 * contains functions for calculating state of charge, including:
 * timer interrupt which triggers ADC conversion
 * ADC interrupt to collect conversion result
 * function to update SoC value with latest ADC data
 */

///////////////includes///////////////////////////////////////////////////////
#include "soc_fns.h"
#include "mcc_generated_files/adc1.h"
#include "mcc_generated_files/tmr1.h"
#include "mcc_generated_files/can_types.h"
#include "mcc_generated_files/pin_manager.h"
#include "eeprom.h"

///////////////defines////////////////////////////////////////////////////////
#define TOTAL_CHARGE_AH                 2.5
#define ADC_REF_VOLT                    3.3
#define ADC_MAX_BITS                    4095
#define FIVE_THREE_V_DIV                9.1 / (4.7 + 9.1)
#define G_V_PER_A_HIGH                  0.004
#define G_V_PER_A_LOW                   0.0267
#define SAMPLE_TIME_S_X_THOU            4
#define OFFSET_VOLTAGE                  2.5065 //TODO is this correct? This seems to be what the ADC is reading on the CS Low pin when no current is flowing

const float CS_SAMPLE_HI_COEFF =        ADC_REF_VOLT * SAMPLE_TIME_S_X_THOU / (ADC_MAX_BITS * FIVE_THREE_V_DIV * G_V_PER_A_HIGH * 60 * 60);
const float OFFSET_HI_COEFF =           SAMPLE_TIME_S_X_THOU / (G_V_PER_A_HIGH * 60 * 60);
const float CS_SAMPLE_LO_COEFF =        ADC_REF_VOLT * SAMPLE_TIME_S_X_THOU / (ADC_MAX_BITS * FIVE_THREE_V_DIV * G_V_PER_A_LOW * 60 * 60);
const float OFFSET_LO_COEFF =           SAMPLE_TIME_S_X_THOU / (G_V_PER_A_LOW * 60 * 60);

#define USE_LO_THRESHOLD                3200 //~60 A in ADC bits as measured by CS LO channel

#define CS_HIGH_ADC_BITS_TO_AMPS(x)     ((x * ADC_REF_VOLT * 10 / (ADC_MAX_BITS * FIVE_THREE_V_DIV * G_V_PER_A_HIGH)) - OFFSET_VOLTAGE/G_V_PER_A_HIGH * 10)
#define CS_LOW_ADC_BITS_TO_AMPS(x)      ((x * ADC_REF_VOLT * 10 / (ADC_MAX_BITS * FIVE_THREE_V_DIV * G_V_PER_A_LOW)) - OFFSET_VOLTAGE/G_V_PER_A_LOW * 100)

///////////////globals////////////////////////////////////////////////////////
int16_t cs_lo_to_transmit = 0;
int16_t cs_hi_to_transmit = 0;
volatile uint16_t cs_low_sample = 0;
volatile uint16_t cs_high_sample = 0;
float state_of_charge_float = 1000;
volatile uint32_t sum_cs_lo_samples = 0;
volatile uint32_t sum_cs_hi_samples = 0;
volatile uint16_t sample_lo_count = 0;
volatile uint16_t sample_hi_count = 0;
uint8_t eeprom_write_counter = 0;

///////////////interrupt prototypes///////////////////////////////////////////
void timer1_interrupt(void);
void adc1_cs_lo_interrupt(uint16_t valCS_LO);

///////////////functions//////////////////////////////////////////////////////
// initialize peripherals necessary for SoC calculation
void soc_initialize(void)
{
    TMR1_SetInterruptHandler(timer1_interrupt); //my function to handle timer1 interrupts
    ADC1_SetCS_LOInterruptHandler(adc1_cs_lo_interrupt); //my function to handle ADC interrupts

    uint16_t state_of_charge_int = get_state_of_charge_from_eeprom();
    state_of_charge_float = (float)state_of_charge_int;

    ADC1_Enable();
    TMR1_Start();
}

// calculate SoC based on most recently collected current data
void calc_soc(void)
{
    //contribution from low channel + contribution from high channel
```

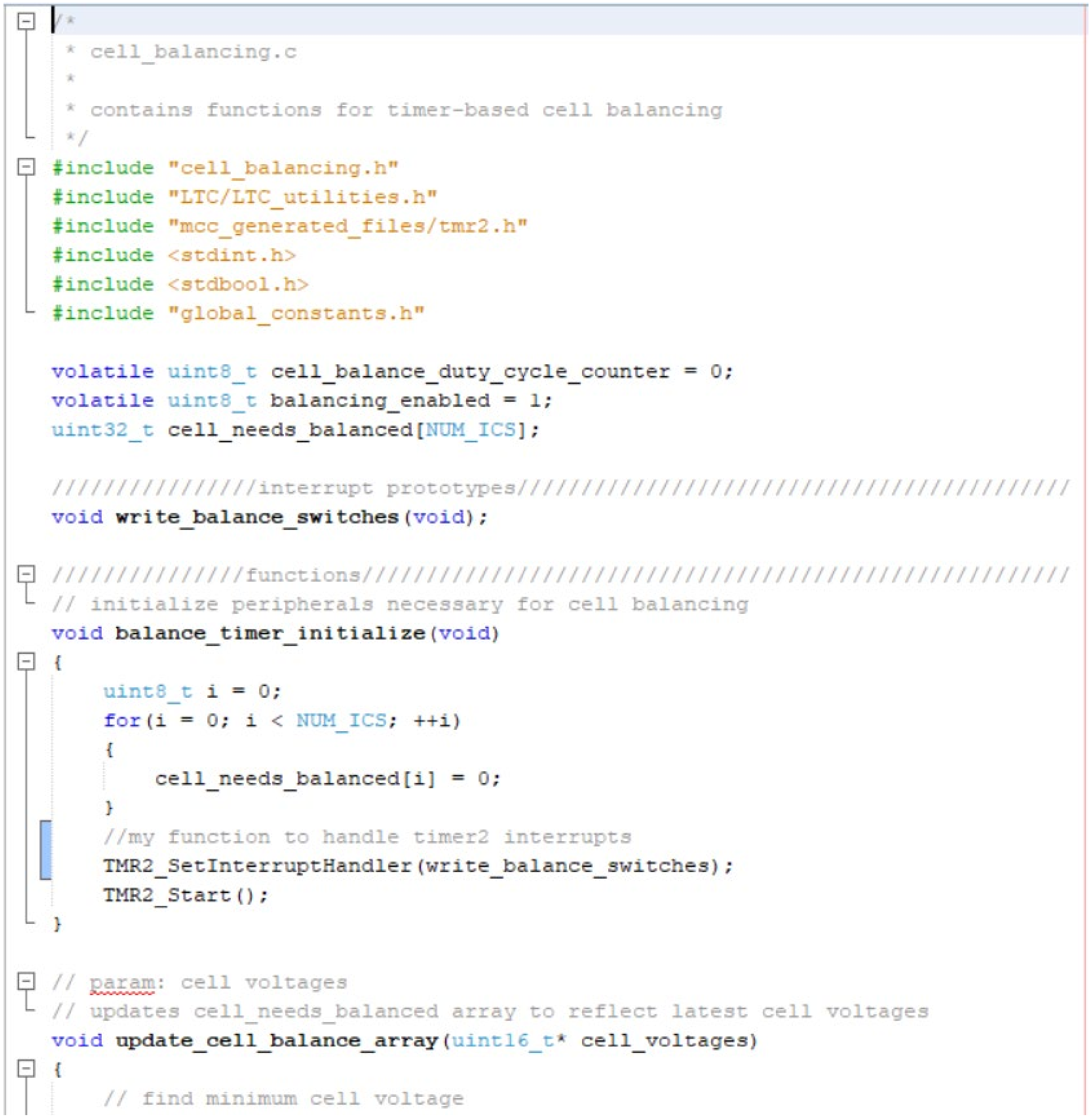


```c
    float spent_Ah_xten = CS_SAMPLE_LO_COEFF * sum_cs_lo_samples - OFFSET_LO_COEFF * OFFSET_VOLTAGE * sample_lo_count;
    spent_Ah_xten += CS_SAMPLE_HI_COEFF * sum_cs_hi_samples - OFFSET_HI_COEFF * OFFSET_VOLTAGE * sample_hi_count;
    //reset running counters since we've handled this data
    sum_cs_lo_samples = 0;
    sample_lo_count = 0;
    sum_cs_hi_samples = 0;
    sample_hi_count = 0;

    state_of_charge_float = state_of_charge_float - spent_Ah_xten / TOTAL_CHARGE_AH;
    //can't go higher than 100% SoC or lower than 0%
    if(state_of_charge_float > 1000)
    {
        state_of_charge_float = 1000;
    }
    else if(state_of_charge_float < 0)
    {
        state_of_charge_float = 0;
    }

    cs_lo_to_transmit = (int16_t)CS_LOW_ADC_BITS_TO_AMPS(ADCBUF2);

    /* TODO: why is cs_hi val always junk?
     * do I need/want the following line? cs_high_sample is often junk value
     * since we only retrieve the high sample val if current is too big to be
     * measured using the low channel
     */
    cs_hi_to_transmit = (int16_t)CS_HIGH_ADC_BITS_TO_AMPS(ADCBUF3);

    ++eeprom_write_counter;
    if(eeprom_write_counter > 20)
    {
        eeprom_write_counter = 0;
        LED5_EEPROM_Toggle();
        uint16_t soc_int = (uint16_t)state_of_charge_float;
        write_eeprom(soc_int);
    }
}
```

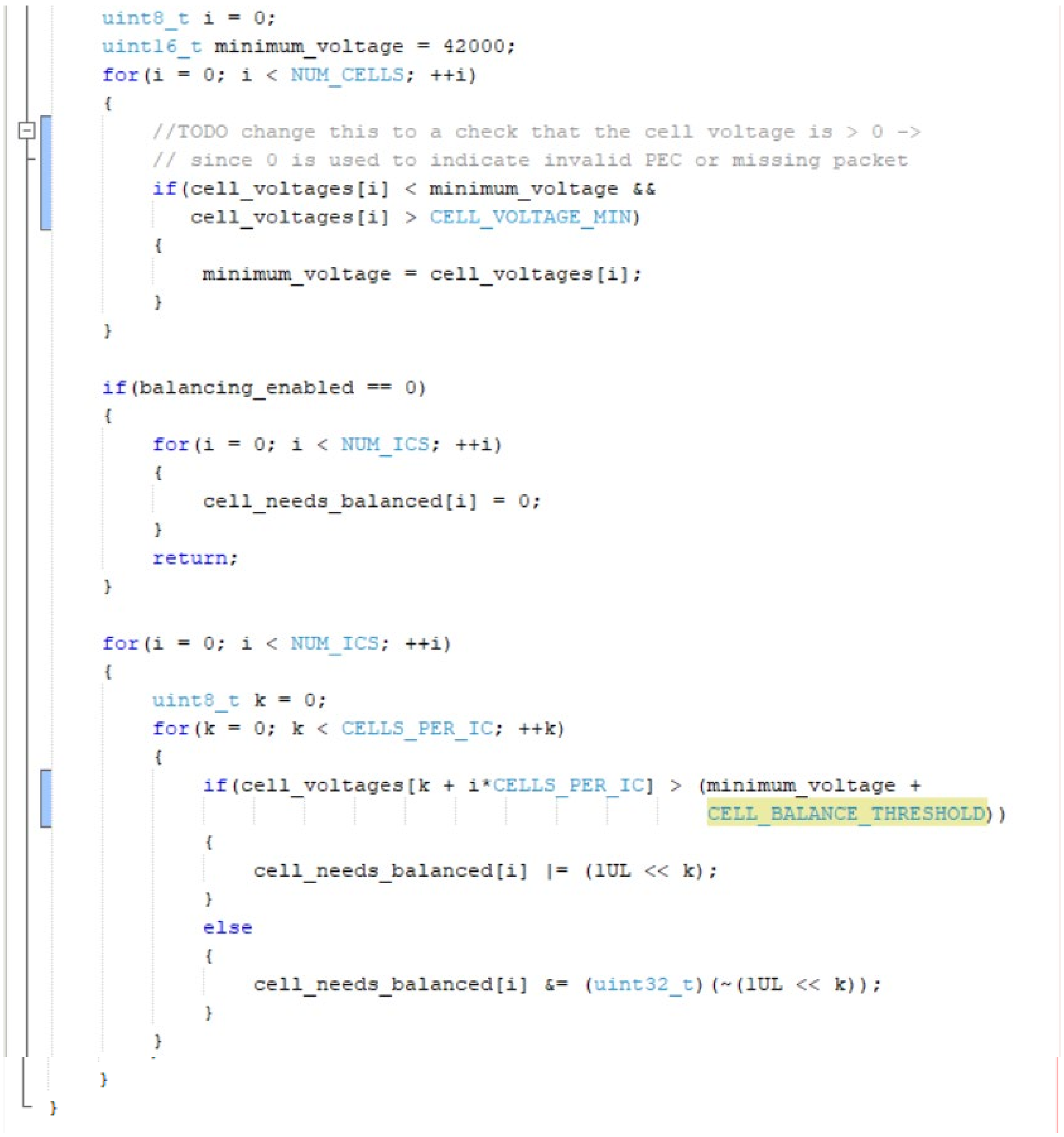


```c
///////////////interrupts/////////////////////////////////////////////////////
//4 ms timer
void timer1_interrupt(void)
{
    LED3_TMR1_SetHigh();

    //start ADC sampling for both channels
    ADC1_SoftwareTriggerEnable();

    LED3_TMR1_SetLow();
}

//interrupt is triggered when sampling is complete
void adc1_cs_lo_interrupt(uint16_t valCS_LO)
{
    LED4_ADC_SetHigh();

    // if current low enough to be measured by low channel
    if(valCS_LO < USE_LO_THRESHOLD)
    {
        //keep tally of ADC samples
        sum_cs_lo_samples = sum_cs_lo_samples + valCS_LO;
        cs_low_sample = valCS_LO;
        sample_lo_count = sample_lo_count + 1;
    }
    else //else need to use high channel
    {
        uint16_t valCS_HI = ADC1_ConversionResultGet(CS_HI);
        sum_cs_hi_samples = sum_cs_hi_samples + valCS_HI;
        cs_high_sample = valCS_HI;
        sample_hi_count = sample_hi_count + 1;
    }

    LED4_ADC_SetLow();
}
```

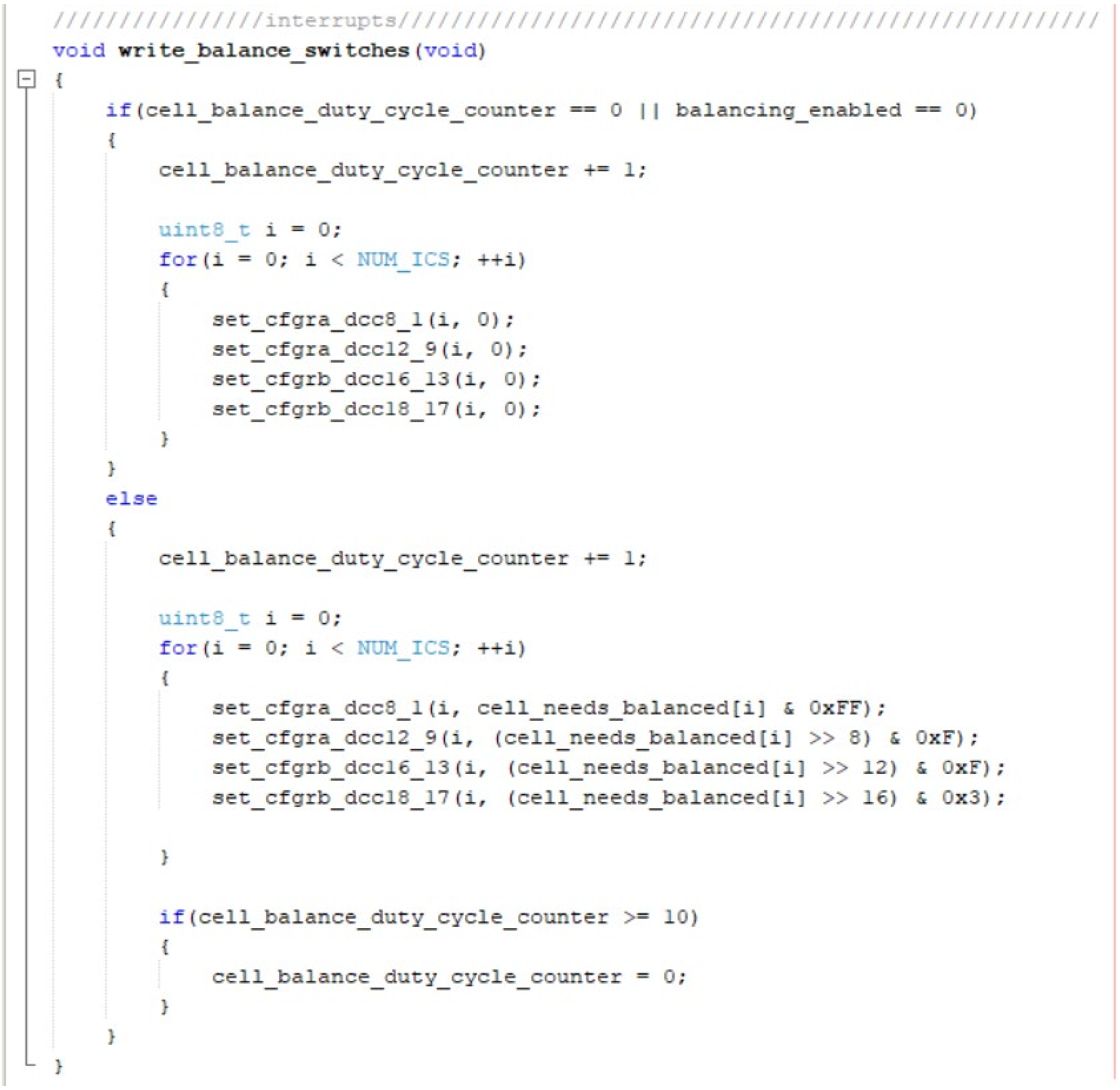
Cell Balancing Algorithm:



```
/*
 * cell_balancing.c
 *
 * contains functions for timer-based cell balancing
 */
#include "cell_balancing.h"
#include "LTC/LTC_utilities.h"
#include "mcc_generated_files/tmr2.h"
#include <stdint.h>
#include <stdbool.h>
#include "global_constants.h"

volatile uint8_t cell_balance_duty_cycle_counter = 0;
volatile uint8_t balancing_enabled = 1;
uint32_t cell_needs_balanced[NUM_ICS];

////////////////interrupt prototypes////////////////////////////////////////////
void write_balance_switches(void);

///////////////functions//////////////////////////////////////////////////////
// initialize peripherals necessary for cell balancing
void balance_timer_initialize(void)
{
    uint8_t i = 0;
    for(i = 0; i < NUM_ICS; ++i)
    {
        cell_needs_balanced[i] = 0;
    }
    //my function to handle timer2 interrupts
    TMR2_SetInterruptHandler(write_balance_switches);
    TMR2_Start();
}

// param: cell voltages
// updates cell_needs_balanced array to reflect latest cell voltages
void update_cell_balance_array(uint16_t* cell_voltages)
{
    // find minimum cell voltage
```

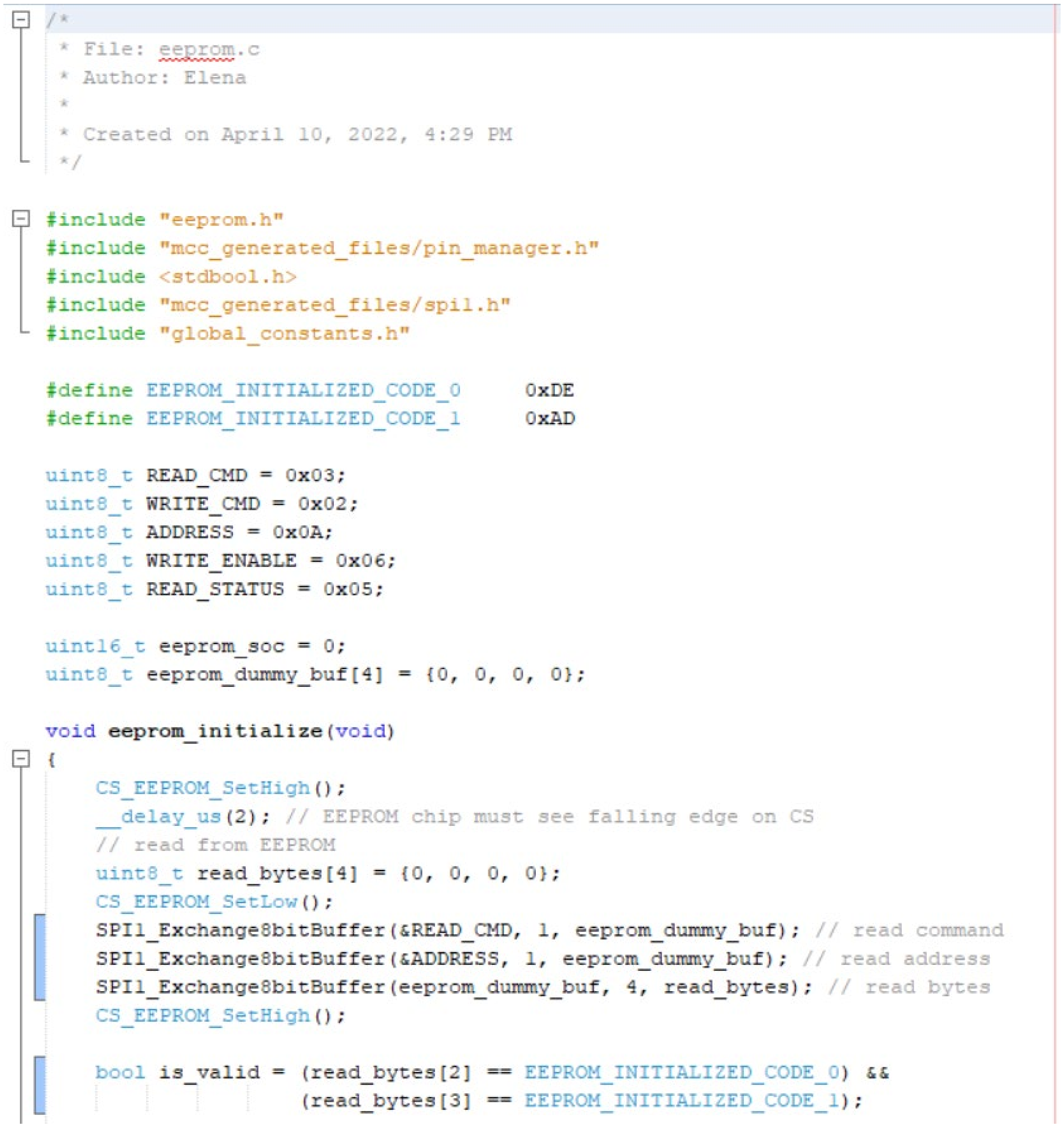


```
    uint8_t i = 0;
    uint16_t minimum_voltage = 42000;
    for(i = 0; i < NUM_CELLS; ++i)
    {
        //TODO change this to a check that the cell voltage is > 0 ->
        // since 0 is used to indicate invalid PEC or missing packet
        if(cell_voltages[i] < minimum_voltage &&
           cell_voltages[i] > CELL_VOLTAGE_MIN)
        {
            minimum_voltage = cell_voltages[i];
        }
    }

    if(balancing_enabled == 0)
    {
        for(i = 0; i < NUM_ICS; ++i)
        {
            cell_needs_balanced[i] = 0;
        }
        return;
    }

    for(i = 0; i < NUM_ICS; ++i)
    {
        uint8_t k = 0;
        for(k = 0; k < CELLS_PER_IC; ++k)
        {
            if(cell_voltages[k + i*CELLS_PER_IC] > (minimum_voltage +
                                                    CELL_BALANCE_THRESHOLD))
            {
                cell_needs_balanced[i] |= (1UL << k);
            }
            else
            {
                cell_needs_balanced[i] &= (uint32_t)(~(1UL << k));
            }
        }
    }
}
```

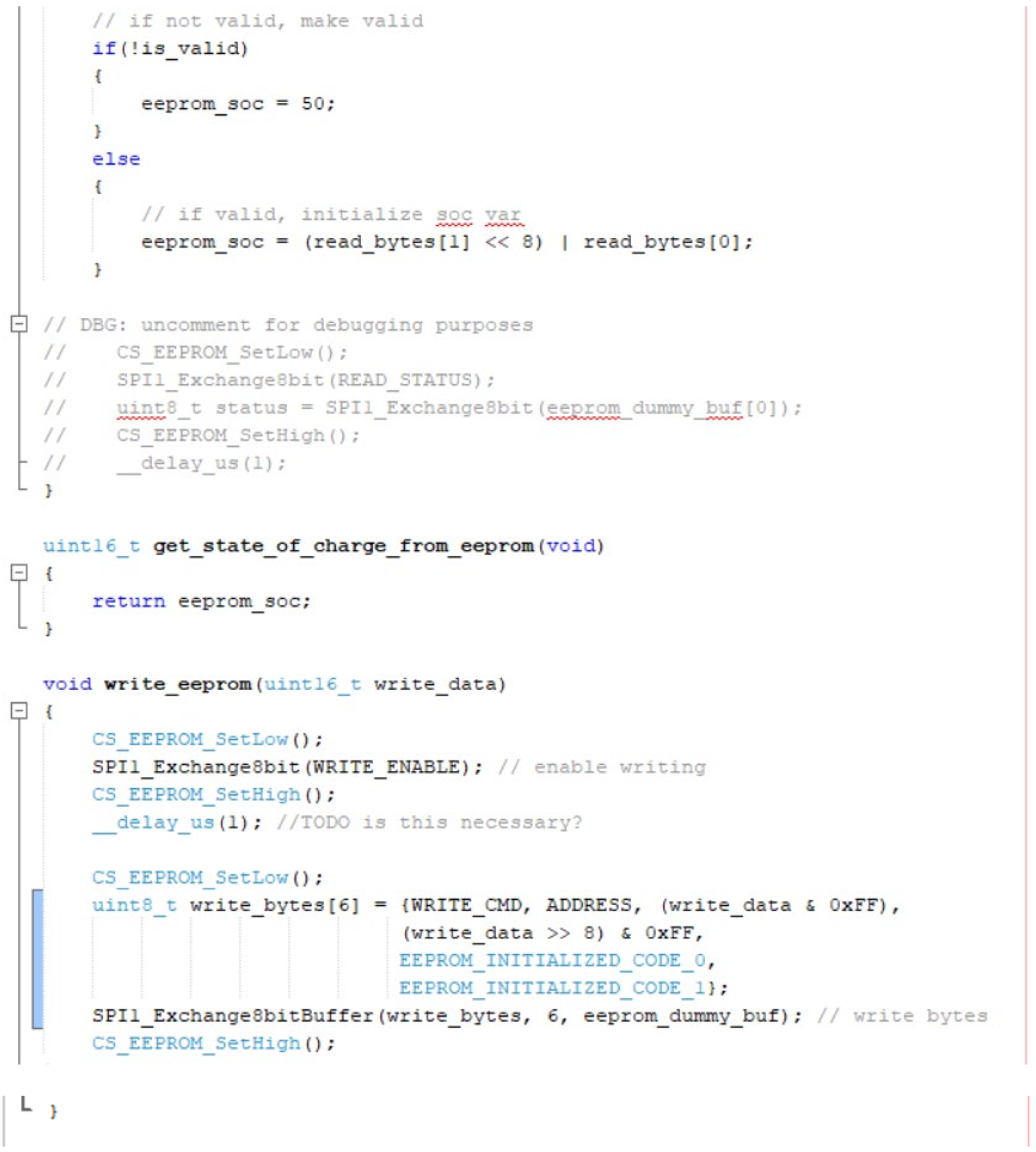


```
///////////////interrupts////////////////////////////////////////////////////
void write_balance_switches(void)
{
    if(cell_balance_duty_cycle_counter == 0 || balancing_enabled == 0)
    {
        cell_balance_duty_cycle_counter += 1;

        uint8_t i = 0;
        for(i = 0; i < NUM_ICS; ++i)
        {
            set_cfgra_dcc8_1(i, 0);
            set_cfgra_dcc12_9(i, 0);
            set_cfgrb_dcc16_13(i, 0);
            set_cfgrb_dcc18_17(i, 0);
        }
    }
    else
    {
        cell_balance_duty_cycle_counter += 1;

        uint8_t i = 0;
        for(i = 0; i < NUM_ICS; ++i)
        {
            set_cfgra_dcc8_1(i, cell_needs_balanced[i] & 0xFF);
            set_cfgra_dcc12_9(i, (cell_needs_balanced[i] >> 8) & 0xF);
            set_cfgrb_dcc16_13(i, (cell_needs_balanced[i] >> 12) & 0xF);
            set_cfgrb_dcc18_17(i, (cell_needs_balanced[i] >> 16) & 0x3);

        }

        if(cell_balance_duty_cycle_counter >= 10)
        {
            cell_balance_duty_cycle_counter = 0;
        }
    }
}
```

EEPROM Functions:

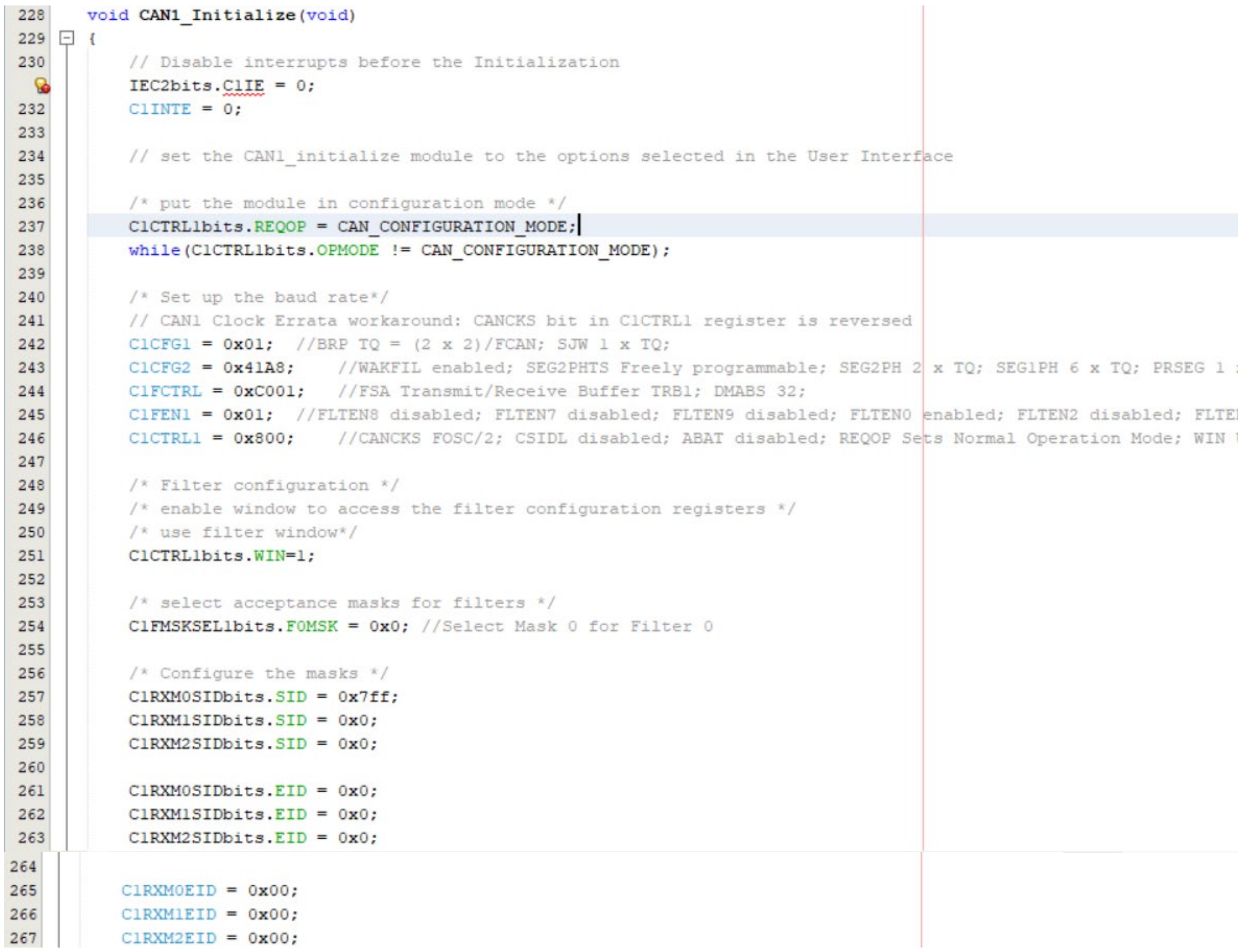


```
/*
 * File: eeprom.c
 * Author: Elena
 *
 * Created on April 10, 2022, 4:29 PM
 */

#include "eeprom.h"
#include "mcc_generated_files/pin_manager.h"
#include <stdbool.h>
#include "mcc_generated_files/spi1.h"
#include "global_constants.h"

#define EEPROM_INITIALIZED_CODE_0     0xDE
#define EEPROM_INITIALIZED_CODE_1     0xAD

uint8_t READ_CMD = 0x03;
uint8_t WRITE_CMD = 0x02;
uint8_t ADDRESS = 0x0A;
uint8_t WRITE_ENABLE = 0x06;
uint8_t READ_STATUS = 0x05;

uint16_t eeprom_soc = 0;
uint8_t eeprom_dummy_buf[4] = {0, 0, 0, 0};

void eeprom_initialize(void)
{
    CS_EEPROM_SetHigh();
    __delay_us(2); // EEPROM chip must see falling edge on CS
    // read from EEPROM
    uint8_t read_bytes[4] = {0, 0, 0, 0};
    CS_EEPROM_SetLow();
    SPI1_Exchange8bitBuffer(&READ_CMD, 1, eeprom_dummy_buf); // read command
    SPI1_Exchange8bitBuffer(&ADDRESS, 1, eeprom_dummy_buf); // read address
    SPI1_Exchange8bitBuffer(eeprom_dummy_buf, 4, read_bytes); // read bytes
    CS_EEPROM_SetHigh();

    bool is_valid = (read_bytes[2] == EEPROM_INITIALIZED_CODE_0) &&
                    (read_bytes[3] == EEPROM_INITIALIZED_CODE_1);
```

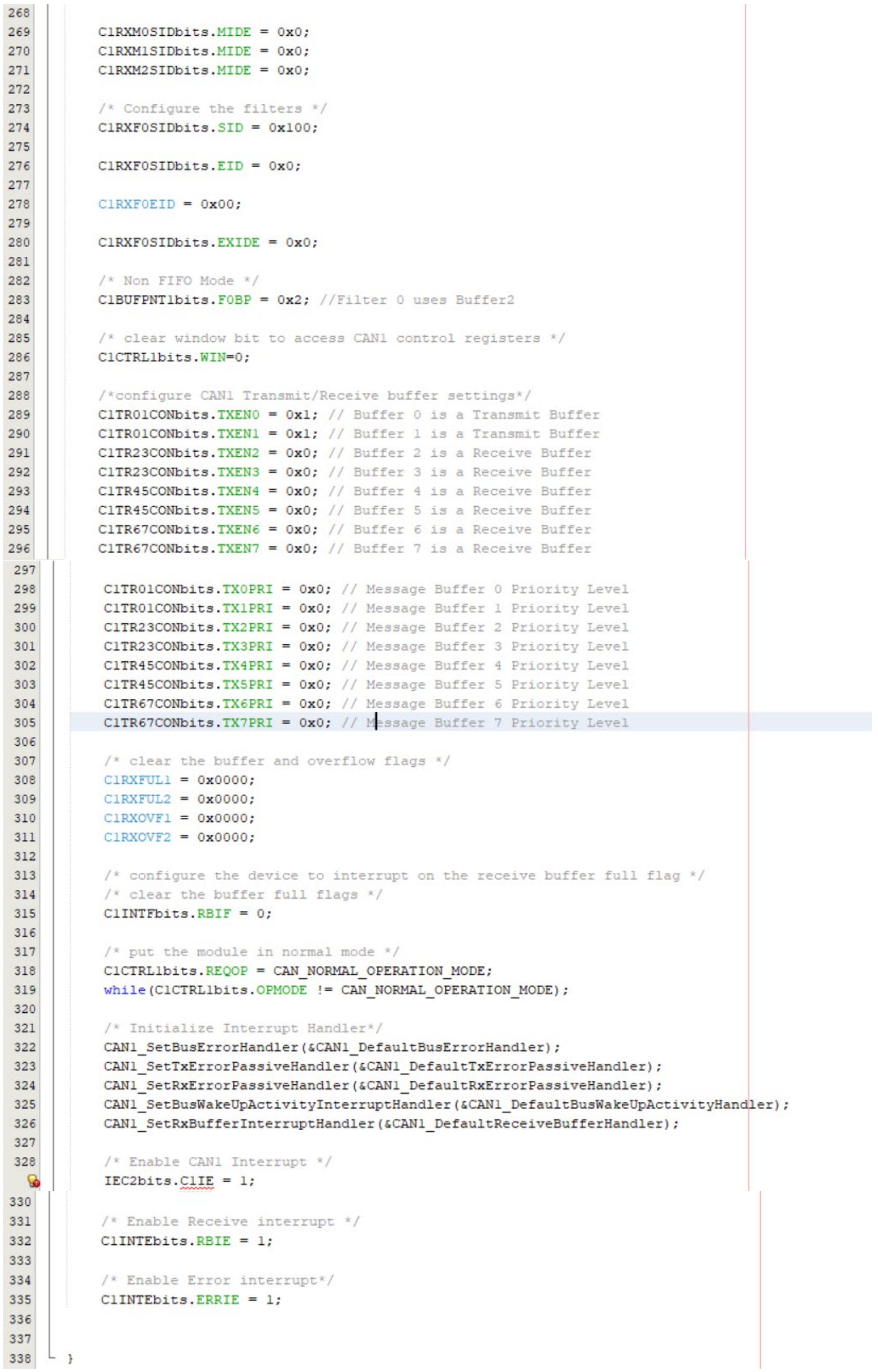


```
        // if not valid, make valid
        if(!is_valid)
        {
            eeprom_soc = 50;
        }
        else
        {
            // if valid, initialize soc var
            eeprom_soc = (read_bytes[1] << 8) | read_bytes[0];
        }

    // DBG: uncomment for debugging purposes
    //     CS_EEPROM_SetLow();
    //     SPI1_Exchange8bit(READ_STATUS);
    //     uint8_t status = SPI1_Exchange8bit(eeprom_dummy_buf[0]);
    //     CS_EEPROM_SetHigh();
    //     __delay_us(1);
    }

    uint16_t get_state_of_charge_from_eeprom(void)
    {
        return eeprom_soc;
    }

    void write_eeprom(uint16_t write_data)
    {
        CS_EEPROM_SetLow();
        SPI1_Exchange8bit(WRITE_ENABLE); // enable writing
        CS_EEPROM_SetHigh();
        __delay_us(1); //TODO is this necessary?

        CS_EEPROM_SetLow();
        uint8_t write_bytes[6] = {WRITE_CMD, ADDRESS, (write_data & 0xFF),
                                  (write_data >> 8) & 0xFF,
                                  EEPROM_INITIALIZED_CODE_0,
                                  EEPROM_INITIALIZED_CODE_1};
        SPI1_Exchange8bitBuffer(write_bytes, 6, eeprom_dummy_buf); // write bytes
        CS_EEPROM_SetHigh();

    }
```

CAN peripheral initialization:



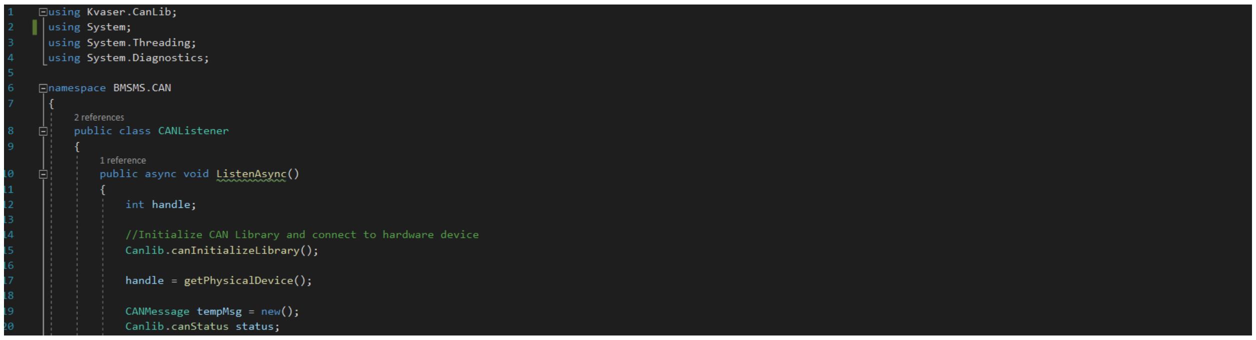
```
void CAN1_Initialize(void)
{
    // Disable interrupts before the Initialization
    IEC2bits.C1IE = 0;
    C1INTE = 0;

    // set the CAN1_initialize module to the options selected in the User Interface

    /* put the module in configuration mode */
    C1CTRL1bits.REQOP = CAN_CONFIGURATION_MODE;
    while(C1CTRL1bits.OPMODE != CAN_CONFIGURATION_MODE);

    /* Set up the baud rate*/
    // CAN1 Clock Errata workaround: CANCKS bit in C1CTRL1 register is reversed
    C1CFG1 = 0x01;  //BRP TQ = (2 x 2)/FCAN; SJW 1 x TQ;
    C1CFG2 = 0x41A8;    //WAKFIL enabled; SEG2PHTS Freely programmable; SEG2PH 2 x TQ; SEG1PH 6 x TQ; PRSEG 1
    C1FCTRL = 0xC001;   //FSA Transmit/Receive Buffer TRB1; DMABS 32;
    C1FEN1 = 0x01;  //FLTEN8 disabled; FLTEN7 disabled; FLTEN9 disabled; FLTEN0 enabled; FLTEN2 disabled; FLTE
    C1CTRL1 = 0x800;    //CANCKS FOSC/2; CSIDL disabled; ABAT disabled; REQOP Sets Normal Operation Mode; WIN

    /* Filter configuration */
    /* enable window to access the filter configuration registers */
    /* use filter window*/
    C1CTRL1bits.WIN=1;

    /* select acceptance masks for filters */
    C1FMSKSEL1bits.F0MSK = 0x0; //Select Mask 0 for Filter 0

    /* Configure the masks */
    C1RXM0SIDbits.SID = 0x7ff;
    C1RXM1SIDbits.SID = 0x0;
    C1RXM2SIDbits.SID = 0x0;

    C1RXM0SIDbits.EID = 0x0;
    C1RXM1SIDbits.EID = 0x0;
    C1RXM2SIDbits.EID = 0x0;

    C1RXM0EID = 0x00;
    C1RXM1EID = 0x00;
    C1RXM2EID = 0x00;
```



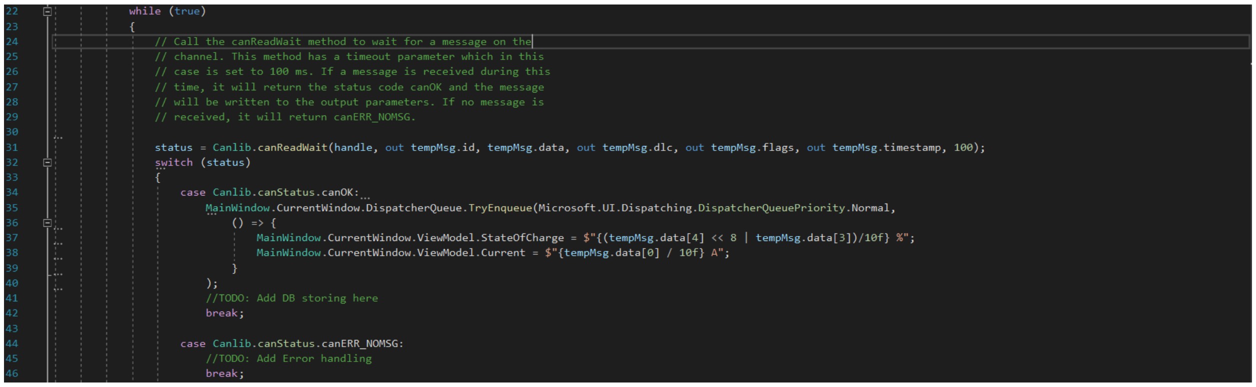
```

    C1RXM0SIDbits.MIDE = 0x0;
    C1RXM1SIDbits.MIDE = 0x0;
    C1RXM2SIDbits.MIDE = 0x0;

    /* Configure the filters */
    C1RXF0SIDbits.SID = 0x100;

    C1RXF0SIDbits.EID = 0x0;

    C1RXF0EID = 0x00;

    C1RXF0SIDbits.EXIDE = 0x0;

    /* Non FIFO Mode */
    C1BUFPNT1bits.F0BP = 0x2; //Filter 0 uses Buffer2

    /* clear window bit to access CAN1 control registers */
    C1CTRL1bits.WIN=0;

    /*configure CAN1 Transmit/Receive buffer settings*/
    C1TR01CONbits.TXEN0 = 0x1; // Buffer 0 is a Transmit Buffer
    C1TR01CONbits.TXEN1 = 0x1; // Buffer 1 is a Transmit Buffer
    C1TR23CONbits.TXEN2 = 0x0; // Buffer 2 is a Receive Buffer
    C1TR23CONbits.TXEN3 = 0x0; // Buffer 3 is a Receive Buffer
    C1TR45CONbits.TXEN4 = 0x0; // Buffer 4 is a Receive Buffer
    C1TR45CONbits.TXEN5 = 0x0; // Buffer 5 is a Receive Buffer
    C1TR67CONbits.TXEN6 = 0x0; // Buffer 6 is a Receive Buffer
    C1TR67CONbits.TXEN7 = 0x0; // Buffer 7 is a Receive Buffer

    C1TR01CONbits.TX0PRI = 0x0; // Message Buffer 0 Priority Level
    C1TR01CONbits.TX1PRI = 0x0; // Message Buffer 1 Priority Level
    C1TR23CONbits.TX2PRI = 0x0; // Message Buffer 2 Priority Level
    C1TR23CONbits.TX3PRI = 0x0; // Message Buffer 3 Priority Level
    C1TR45CONbits.TX4PRI = 0x0; // Message Buffer 4 Priority Level
    C1TR45CONbits.TX5PRI = 0x0; // Message Buffer 5 Priority Level
    C1TR67CONbits.TX6PRI = 0x0; // Message Buffer 6 Priority Level
    C1TR67CONbits.TX7PRI = 0x0; // Message Buffer 7 Priority Level

    /* clear the buffer and overflow flags */
    C1RXFUL1 = 0x0000;
    C1RXFUL2 = 0x0000;
    C1RXOVF1 = 0x0000;
    C1RXOVF2 = 0x0000;

    /* configure the device to interrupt on the receive buffer full flag */
    /* clear the buffer full flags */
    C1INTFbits.RBIF = 0;

    /* put the module in normal mode */
    C1CTRL1bits.REQOP = CAN_NORMAL_OPERATION_MODE;
    while(C1CTRL1bits.OPMODE != CAN_NORMAL_OPERATION_MODE);

    /* Initialize Interrupt Handler*/
    CAN1_SetBusErrorHandler(&CAN1_DefaultBusErrorHandler);
    CAN1_SetTxErrorPassiveHandler(&CAN1_DefaultTxErrorPassiveHandler);
    CAN1_SetRxErrorPassiveHandler(&CAN1_DefaultRxErrorPassiveHandler);
    CAN1_SetBusWakeUpActivityInterruptHandler(&CAN1_DefaultBusWakeUpActivityHandler);
    CAN1_SetRxBufferInterruptHandler(&CAN1_DefaultReceiveBufferHandler);

    /* Enable CAN1 Interrupt */
    IEC2bits.C1IE = 1;

    /* Enable Receive interrupt */
    C1INTEbits.RBIE = 1;

    /* Enable Error interrupt*/
    C1INTEbits.ERRIE = 1;


}
```

BMSMS Code:

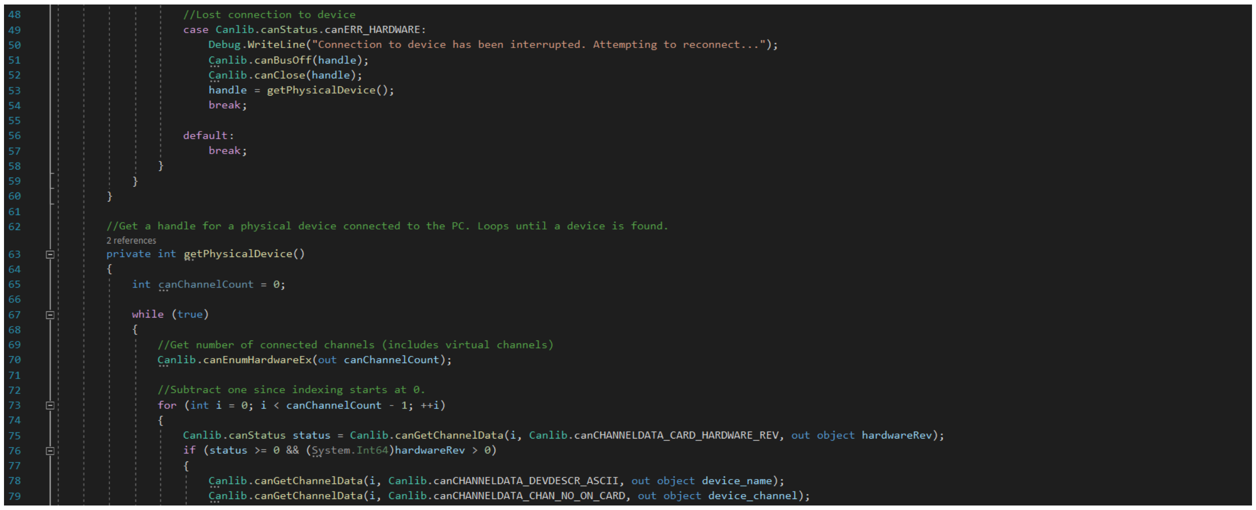
CAN Listener thread:



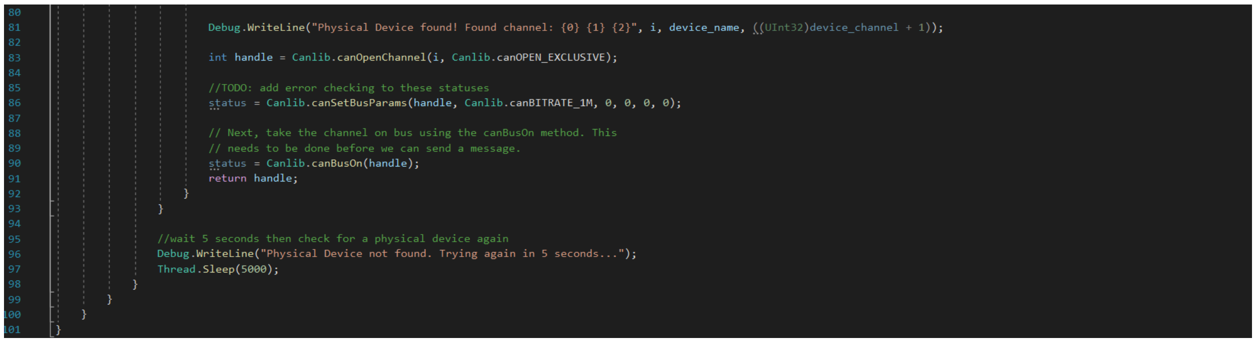
```csharp
using Kvaser.CanLib;
using System;
using System.Threading;
using System.Diagnostics;

namespace BMSMS.CAN
{
    2 references
    public class CANListener
    {
        1 reference
        public async void ListenAsync()
        {
            int handle;

            //Initialize CAN Library and connect to hardware device
            Canlib.canInitializeLibrary();

            handle = getPhysicalDevice();

            CANMessage tempMsg = new();
            Canlib.canStatus status;
```



```csharp
            while (true)
            {
                // Call the canReadWait method to wait for a message on the
                // channel. This method has a timeout parameter which in this
                // case is set to 100 ms. If a message is received during this
                // time, it will return the status code canOK and the message
                // will be written to the output parameters. If no message is
                // received, it will return canERR_NOMSG.

                status = Canlib.canReadWait(handle, out tempMsg.id, tempMsg.data, out tempMsg.dlc, out tempMsg.flags, out tempMsg.timestamp, 100);
                switch (status)
                {
                    case Canlib.canStatus.canOK:
                        MainWindow.CurrentWindow.DispatcherQueue.TryEnqueue(Microsoft.UI.Dispatching.DispatcherQueuePriority.Normal,
                            () => {
                                MainWindow.CurrentWindow.ViewModel.StateOfCharge = $"{(tempMsg.data[4] << 8 | tempMsg.data[3])/10f} %";
                                MainWindow.CurrentWindow.ViewModel.Current = $"{tempMsg.data[0] / 10f} A";
                            }
                        );
                        //TODO: Add DB storing here
                        break;

                    case Canlib.canStatus.canERR_NOMSG:
                        //TODO: Add Error handling
                        break;
```

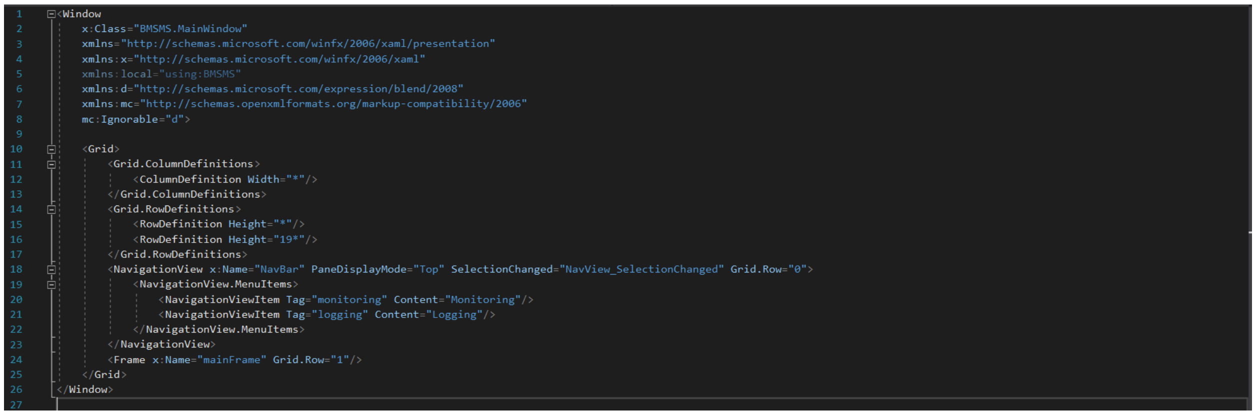


```csharp
                    //Lost connection to device
                    case Canlib.canStatus.canERR_HARDWARE:
                        Debug.WriteLine("Connection to device has been interrupted. Attempting to reconnect...");
                        Canlib.canBusOff(handle);
                        Canlib.canClose(handle);
                        handle = getPhysicalDevice();
                        break;

                    default:
                        break;
                }
            }
        }

        //Get a handle for a physical device connected to the PC. Loops until a device is found.
        2 references
        private int getPhysicalDevice()
        {
            int canChannelCount = 0;

            while (true)
            {
                //Get number of connected channels (includes virtual channels)
                Canlib.canEnumHardwareEx(out canChannelCount);

                //Subtract one since indexing starts at 0.
                for (int i = 0; i < canChannelCount - 1; ++i)
                {
                    Canlib.canStatus status = Canlib.canGetChannelData(i, Canlib.canCHANNELDATA_CARD_HARDWARE_REV, out object hardwareRev);
                    if (status >= 0 && (System.Int64)hardwareRev > 0)
                    {
                        Canlib.canGetChannelData(i, Canlib.canCHANNELDATA_DEVDESCR_ASCII, out object device_name);
                        Canlib.canGetChannelData(i, Canlib.canCHANNELDATA_CHAN_NO_ON_CARD, out object device_channel);
```



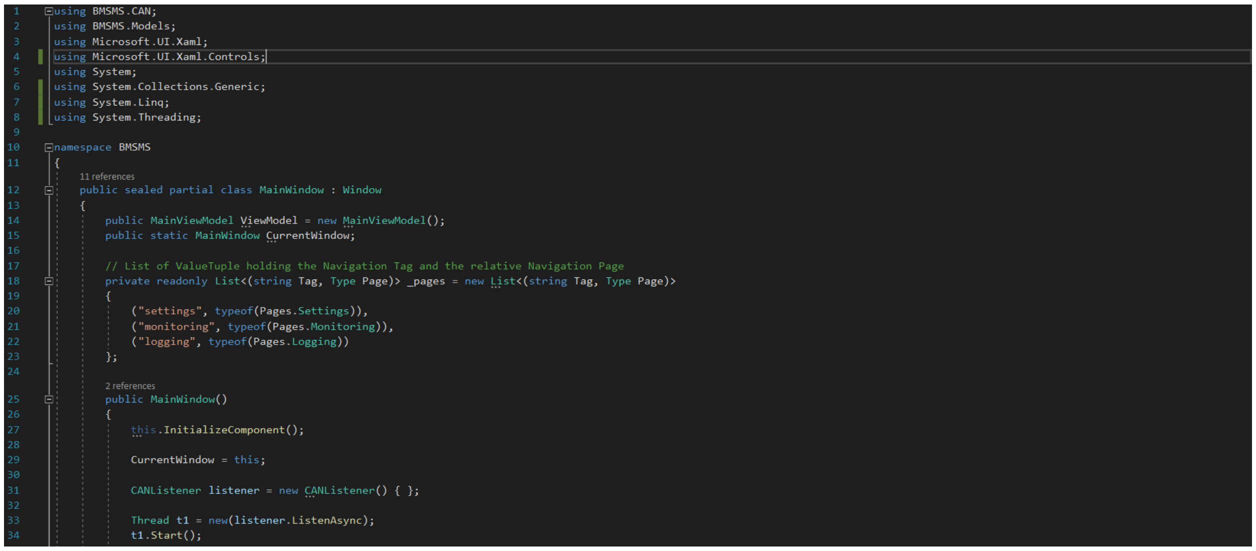
```csharp
                        Debug.WriteLine("Physical Device found! Found channel: {0} {1} {2}", i, device_name, ((UInt32)device_channel + 1));

                        int handle = Canlib.canOpenChannel(i, Canlib.canOPEN_EXCLUSIVE);

                        //TODO: add error checking to these statuses
                        status = Canlib.canSetBusParams(handle, Canlib.canBITRATE_1M, 0, 0, 0, 0);

                        // Next, take the channel on bus using the canBusOn method. This
                        // needs to be done before we can send a message.
                        status = Canlib.canBusOn(handle);
                        return handle;
                    }
                }

                //wait 5 seconds then check for a physical device again
                Debug.WriteLine("Physical Device not found. Trying again in 5 seconds...");
                Thread.Sleep(5000);
            }
        }
    }
}
```

Main Window View:

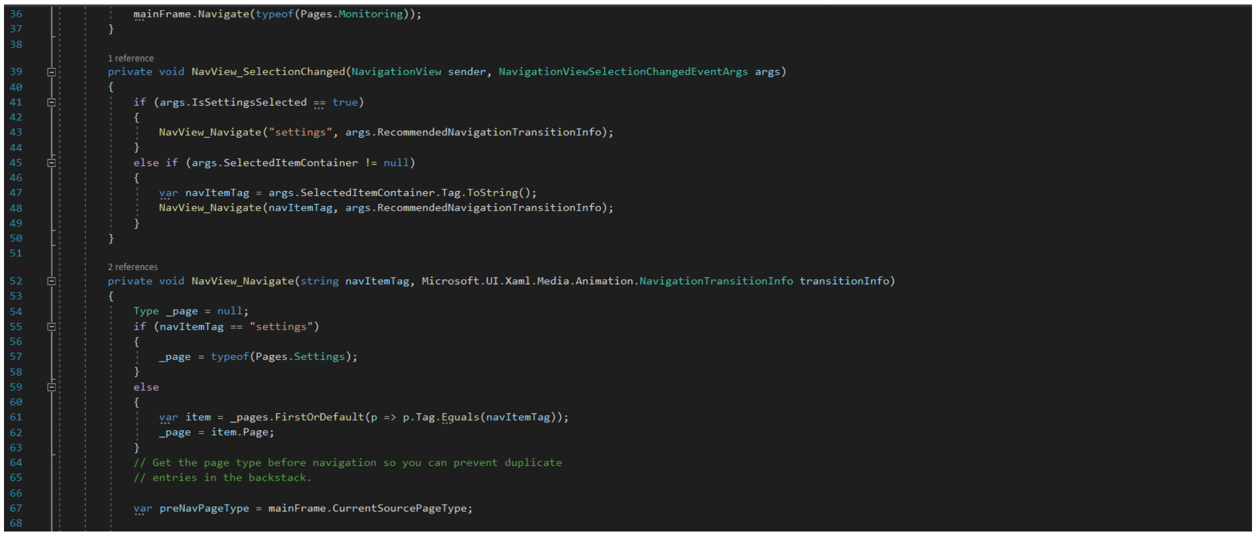


```xml
<Window
    x:Class="BMSMS.MainWindow"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    xmlns:local="using:BMSMS"
    xmlns:d="http://schemas.microsoft.com/expression/blend/2008"
    xmlns:mc="http://schemas.openxmlformats.org/markup-compatibility/2006"
    mc:Ignorable="d">

    <Grid>
        <Grid.ColumnDefinitions>
            <ColumnDefinition Width="*"/>
        </Grid.ColumnDefinitions>
        <Grid.RowDefinitions>
            <RowDefinition Height="*"/>
            <RowDefinition Height="19*"/>
        </Grid.RowDefinitions>
        <NavigationView x:Name="NavBar" PaneDisplayMode="Top" SelectionChanged="NavView_SelectionChanged" Grid.Row="0">
            <NavigationView.MenuItems>
                <NavigationViewItem Tag="monitoring" Content="Monitoring"/>
                <NavigationViewItem Tag="logging" Content="Logging"/>
            </NavigationView.MenuItems>
        </NavigationView>
        <Frame x:Name="mainFrame" Grid.Row="1"/>
    </Grid>
</Window>
```

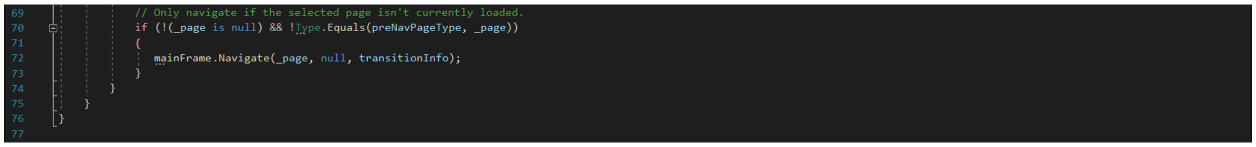
Main Window Code Behind:



```csharp
using BMSMS.CAN;
using BMSMS.Models;
using Microsoft.UI.Xaml;
using Microsoft.UI.Xaml.Controls;
using System;
using System.Collections.Generic;
using System.Linq;
using System.Threading;

namespace BMSMS
{
    11 references
    public sealed partial class MainWindow : Window
    {
        public MainViewModel ViewModel = new MainViewModel();
        public static MainWindow CurrentWindow;

        // List of ValueTuple holding the Navigation Tag and the relative Navigation Page
        private readonly List<(string Tag, Type Page)> _pages = new List<(string Tag, Type Page)>
        {
            ("settings", typeof(Pages.Settings)),
            ("monitoring", typeof(Pages.Monitoring)),
            ("logging", typeof(Pages.Logging))
        };

        2 references
        public MainWindow()
        {
            this.InitializeComponent();

            CurrentWindow = this;

            CANListener listener = new CANListener() { };

            Thread t1 = new(listener.ListenAsync);
            t1.Start();
```

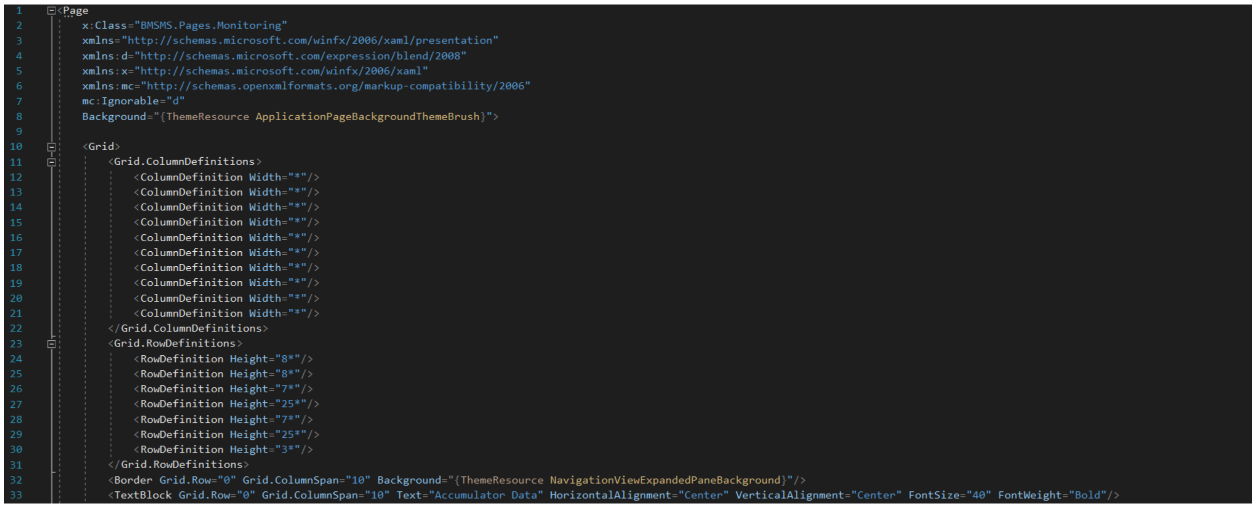


```
            mainFrame.Navigate(typeof(Pages.Monitoring));
        }

        1 reference
        private void NavView_SelectionChanged(NavigationView sender, NavigationViewSelectionChangedEventArgs args)
        {
            if (args.IsSettingsSelected == true)
            {
                NavView_Navigate("settings", args.RecommendedNavigationTransitionInfo);
            }
            else if (args.SelectedItemContainer != null)
            {
                var navItemTag = args.SelectedItemContainer.Tag.ToString();
                NavView_Navigate(navItemTag, args.RecommendedNavigationTransitionInfo);
            }
        }

        2 references
        private void NavView_Navigate(string navItemTag, Microsoft.UI.Xaml.Media.Animation.NavigationTransitionInfo transitionInfo)
        {
            Type _page = null;
            if (navItemTag == "settings")
            {
                _page = typeof(Pages.Settings);
            }
            else
            {
                var item = _pages.FirstOrDefault(p => p.Tag.Equals(navItemTag));
                _page = item.Page;
            }
            // Get the page type before navigation so you can prevent duplicate
            // entries in the backstack.

            var preNavPageType = mainFrame.CurrentSourcePageType;

```

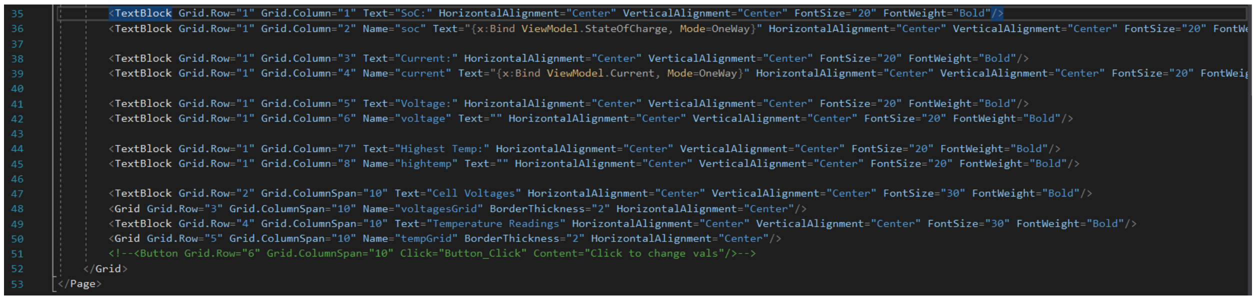


```
            // Only navigate if the selected page isn't currently loaded.
            if (!(_page is null) && !Type.Equals(preNavPageType, _page))
            {
                mainFrame.Navigate(_page, null, transitionInfo);
            }
        }
    }
}

```

Monitoring Page View:

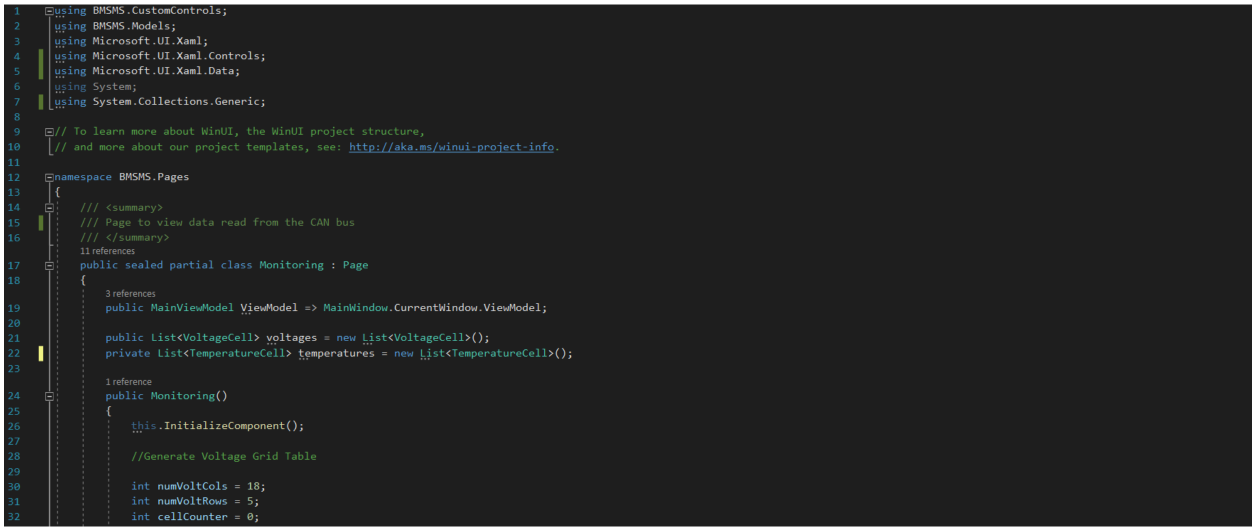


```
<Page
    x:Class="BMSMS.Pages.Monitoring"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:d="http://schemas.microsoft.com/expression/blend/2008"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    xmlns:mc="http://schemas.openxmlformats.org/markup-compatibility/2006"
    mc:Ignorable="d"
    Background="{ThemeResource ApplicationPageBackgroundThemeBrush}">

    <Grid>
        <Grid.ColumnDefinitions>
            <ColumnDefinition Width="*"/>
            <ColumnDefinition Width="*"/>
            <ColumnDefinition Width="*"/>
            <ColumnDefinition Width="*"/>
            <ColumnDefinition Width="*"/>
            <ColumnDefinition Width="*"/>
            <ColumnDefinition Width="*"/>
            <ColumnDefinition Width="*"/>
            <ColumnDefinition Width="*"/>
            <ColumnDefinition Width="*"/>
        </Grid.ColumnDefinitions>
        <Grid.RowDefinitions>
            <RowDefinition Height="8*"/>
            <RowDefinition Height="8*"/>
            <RowDefinition Height="7*"/>
            <RowDefinition Height="25*"/>
            <RowDefinition Height="7*"/>
            <RowDefinition Height="25*"/>
            <RowDefinition Height="3*"/>
        </Grid.RowDefinitions>
        <Border Grid.Row="0" Grid.ColumnSpan="10" Background="{ThemeResource NavigationViewExpandedPaneBackground}"/>
        <TextBlock Grid.Row="0" Grid.ColumnSpan="10" Text="Accumulator Data" HorizontalAlignment="Center" VerticalAlignment="Center" FontSize="40" FontWeight="Bold"/>
```

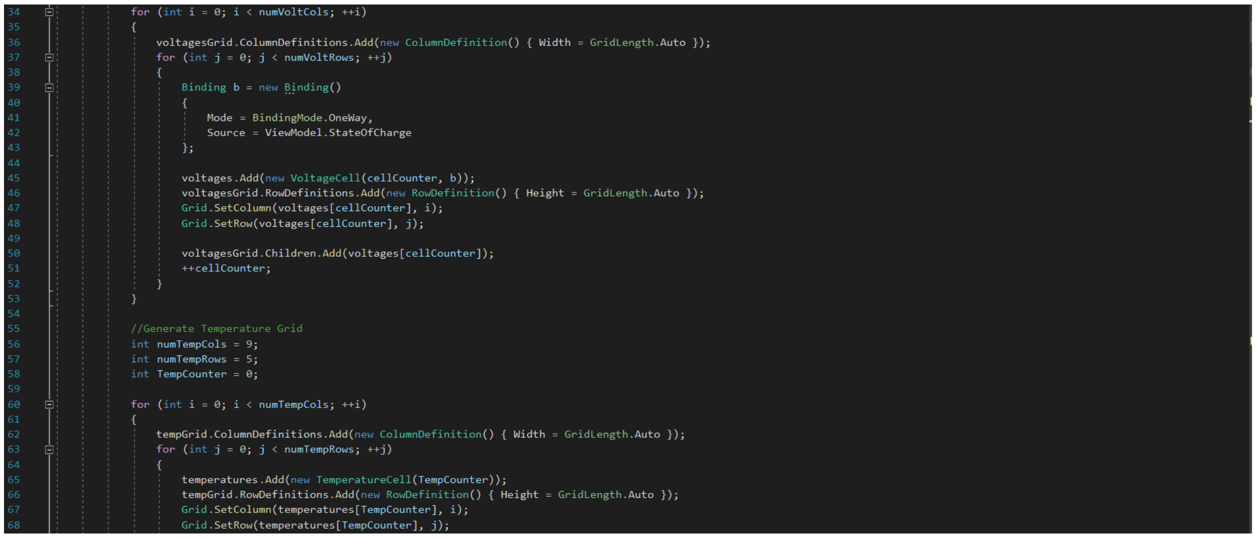


```
        <TextBlock Grid.Row="1" Grid.Column="1" Text="SoC:" HorizontalAlignment="Center" VerticalAlignment="Center" FontSize="20" FontWeight="Bold"/>
        <TextBlock Grid.Row="1" Grid.Column="2" Name="soc" Text="{x:Bind ViewModel.StateOfCharge, Mode=OneWay}" HorizontalAlignment="Center" VerticalAlignment="Center" FontSize="20" FontWe

        <TextBlock Grid.Row="1" Grid.Column="3" Text="Current:" HorizontalAlignment="Center" VerticalAlignment="Center" FontSize="20" FontWeight="Bold"/>
        <TextBlock Grid.Row="1" Grid.Column="4" Name="current" Text="{x:Bind ViewModel.Current, Mode=OneWay}" HorizontalAlignment="Center" VerticalAlignment="Center" FontSize="20" FontWei

        <TextBlock Grid.Row="1" Grid.Column="5" Text="Voltage:" HorizontalAlignment="Center" VerticalAlignment="Center" FontSize="20" FontWeight="Bold"/>
        <TextBlock Grid.Row="1" Grid.Column="6" Name="voltage" Text="" HorizontalAlignment="Center" VerticalAlignment="Center" FontSize="20" FontWeight="Bold"/>

        <TextBlock Grid.Row="1" Grid.Column="7" Text="Highest Temp:" HorizontalAlignment="Center" VerticalAlignment="Center" FontSize="20" FontWeight="Bold"/>
        <TextBlock Grid.Row="1" Grid.Column="8" Name="hightemp" Text="" HorizontalAlignment="Center" VerticalAlignment="Center" FontSize="20" FontWeight="Bold"/>

        <TextBlock Grid.Row="2" Grid.ColumnSpan="10" Text="Cell Voltages" HorizontalAlignment="Center" VerticalAlignment="Center" FontSize="30" FontWeight="Bold"/>
        <Grid Grid.Row="3" Grid.ColumnSpan="10" Name="voltagesGrid" BorderThickness="2" HorizontalAlignment="Center"/>
        <TextBlock Grid.Row="4" Grid.ColumnSpan="10" Text="Temperature Readings" HorizontalAlignment="Center" VerticalAlignment="Center" FontSize="30" FontWeight="Bold"/>
        <Grid Grid.Row="5" Grid.ColumnSpan="10" Name="tempGrid" BorderThickness="2" HorizontalAlignment="Center"/>
        <!--<Button Grid.Row="6" Grid.ColumnSpan="10" Click="Button_Click" Content="Click to change vals"/>-->
    </Grid>
</Page>
```

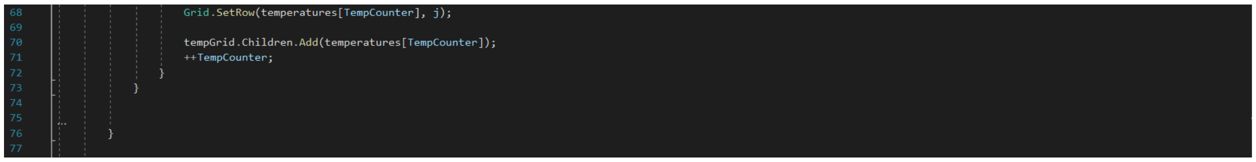
Monitoring Page Code Behind:



```csharp
using BMSMS.CustomControls;
using BMSMS.Models;
using Microsoft.UI.Xaml;
using Microsoft.UI.Xaml.Controls;
using Microsoft.UI.Xaml.Data;
using System;
using System.Collections.Generic;

// To learn more about WinUI, the WinUI project structure,
// and more about our project templates, see: http://aka.ms/winui-project-info.

namespace BMSMS.Pages
{
    /// <summary>
    /// Page to view data read from the CAN bus
    /// </summary>
    11 references
    public sealed partial class Monitoring : Page
    {
        3 references
        public MainViewModel ViewModel => MainWindow.CurrentWindow.ViewModel;

        public List<VoltageCell> voltages = new List<VoltageCell>();
        private List<TemperatureCell> temperatures = new List<TemperatureCell>();

        1 reference
        public Monitoring()
        {
            this.InitializeComponent();

            //Generate Voltage Grid Table

            int numVoltCols = 18;
            int numVoltRows = 5;
            int cellCounter = 0;
```

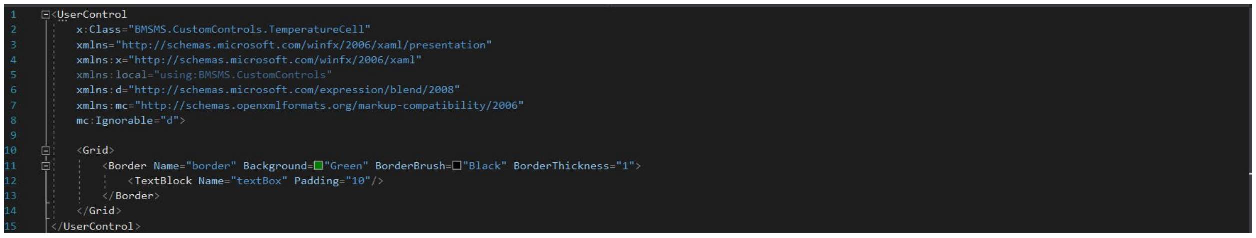


```csharp
            for (int i = 0; i < numVoltCols; ++i)
            {
                voltagesGrid.ColumnDefinitions.Add(new ColumnDefinition() { Width = GridLength.Auto });
                for (int j = 0; j < numVoltRows; ++j)
                {
                    Binding b = new Binding()
                    {
                        Mode = BindingMode.OneWay,
                        Source = ViewModel.StateOfCharge
                    };

                    voltages.Add(new VoltageCell(cellCounter, b));
                    voltagesGrid.RowDefinitions.Add(new RowDefinition() { Height = GridLength.Auto });
                    Grid.SetColumn(voltages[cellCounter], i);
                    Grid.SetRow(voltages[cellCounter], j);

                    voltagesGrid.Children.Add(voltages[cellCounter]);
                    ++cellCounter;
                }
            }

            //Generate Temperature Grid
            int numTempCols = 9;
            int numTempRows = 5;
            int TempCounter = 0;

            for (int i = 0; i < numTempCols; ++i)
            {
                tempGrid.ColumnDefinitions.Add(new ColumnDefinition() { Width = GridLength.Auto });
                for (int j = 0; j < numTempRows; ++j)
                {
                    temperatures.Add(new TemperatureCell(TempCounter));
                    tempGrid.RowDefinitions.Add(new RowDefinition() { Height = GridLength.Auto });
                    Grid.SetColumn(temperatures[TempCounter], i);
                    Grid.SetRow(temperatures[TempCounter], j);
```



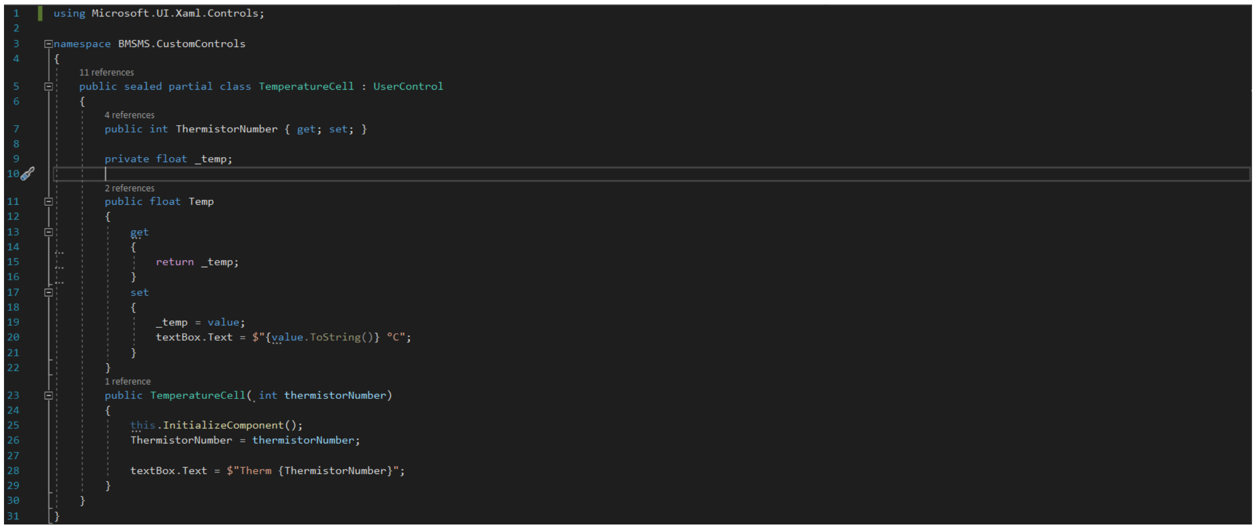
```csharp
                    Grid.SetRow(temperatures[TempCounter], j);

                    tempGrid.Children.Add(temperatures[TempCounter]);
                    ++TempCounter;
                }
            }

 ...
        }

```

Temperature Cell View:



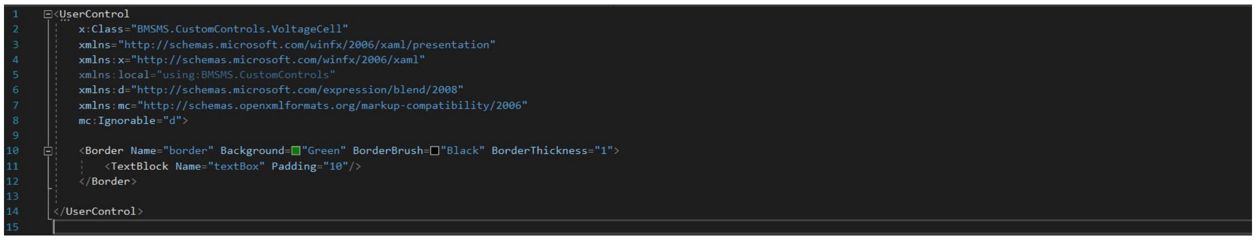
```xml
<UserControl
    x:Class="BMSMS.CustomControls.TemperatureCell"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    xmlns:local="using:BMSMS.CustomControls"
    xmlns:d="http://schemas.microsoft.com/expression/blend/2008"
    xmlns:mc="http://schemas.openxmlformats.org/markup-compatibility/2006"
    mc:Ignorable="d">

    <Grid>
        <Border Name="border" Background=🟩"Green" BorderBrush=□"Black" BorderThickness="1">
            <TextBlock Name="textBox" Padding="10"/>
        </Border>
    </Grid>
</UserControl>
```

Temperature Cell Code Behind:



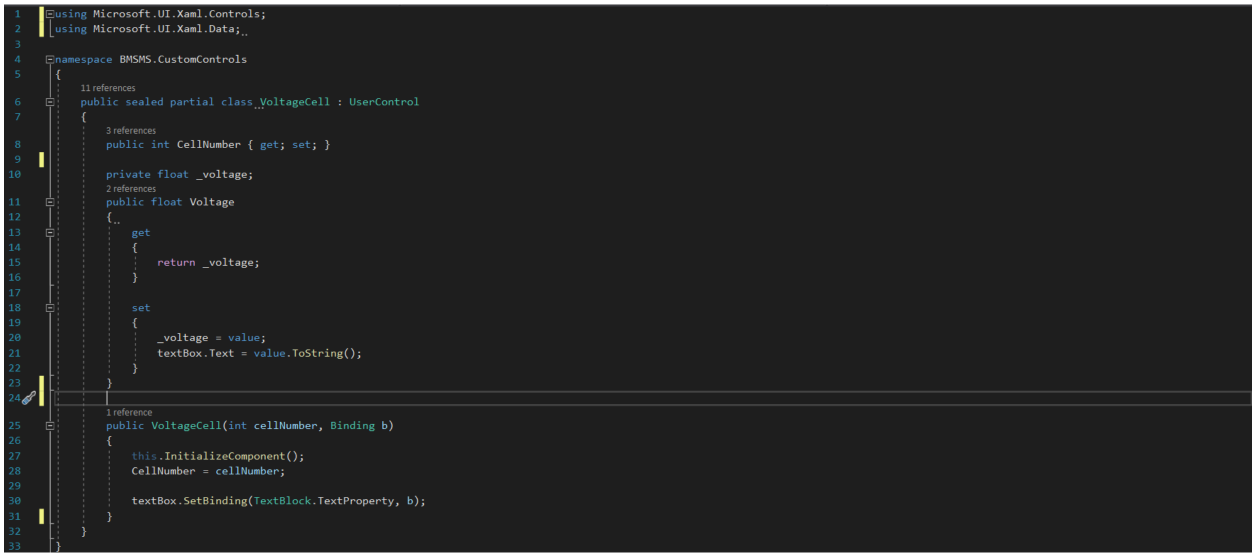
```
using Microsoft.UI.Xaml.Controls;

namespace BMSMS.CustomControls
{
    11 references
    public sealed partial class TemperatureCell : UserControl
    {
        4 references
        public int ThermistorNumber { get; set; }

        private float _temp;

        2 references
        public float Temp
        {
            get
            {
                return _temp;
            }
            set
            {
                _temp = value;
                textBox.Text = $"{value.ToString()} °C";
            }
        }
        1 reference
        public TemperatureCell( int thermistorNumber)
        {
            this.InitializeComponent();
            ThermistorNumber = thermistorNumber;

            textBox.Text = $"Therm {ThermistorNumber}";
        }
    }
}
```

Voltage Cell View:



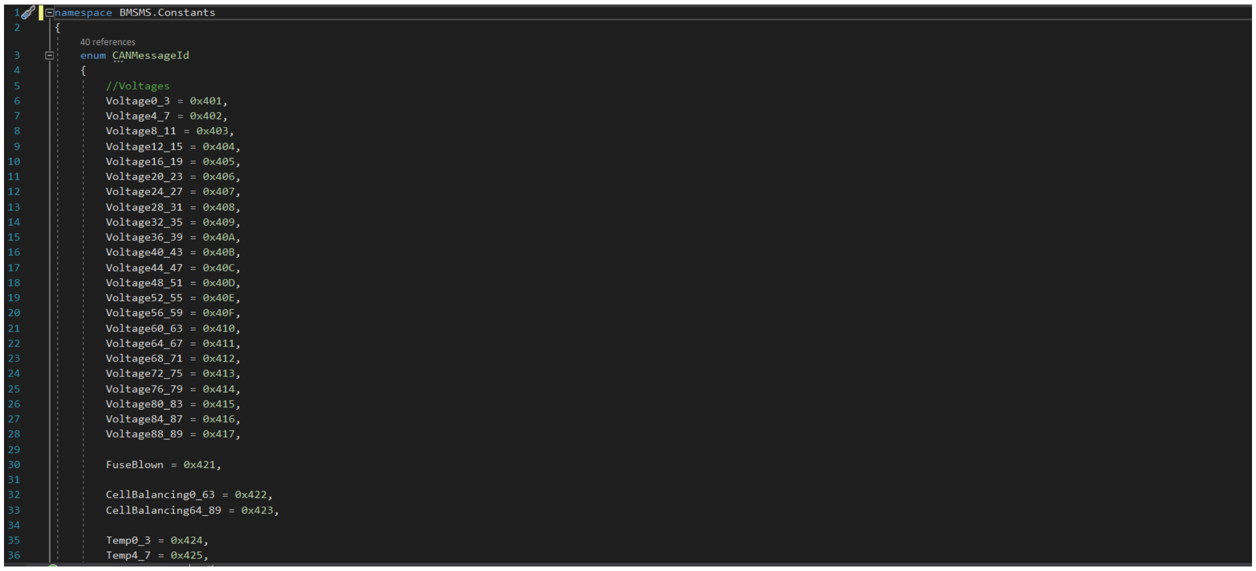
```
<UserControl
    x:Class="BMSMS.CustomControls.VoltageCell"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    xmlns:local="using:BMSMS.CustomControls"
    xmlns:d="http://schemas.microsoft.com/expression/blend/2008"
    xmlns:mc="http://schemas.openxmlformats.org/markup-compatibility/2006"
    mc:Ignorable="d">

    <Border Name="border" Background=🟩"Green" BorderBrush=■"Black" BorderThickness="1">
        <TextBlock Name="textBox" Padding="10"/>
    </Border>

</UserControl>

```

Voltage Cell Code Behind:

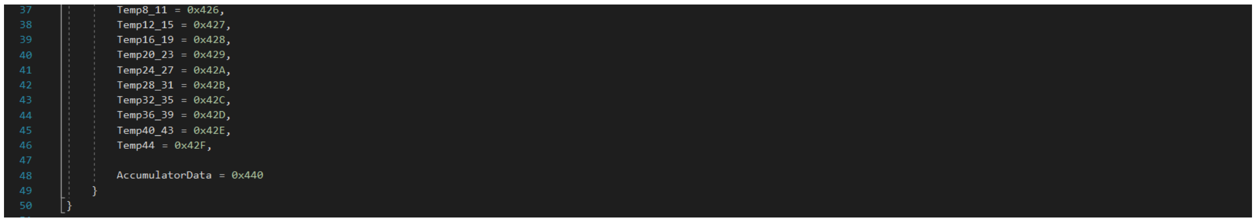


```
using Microsoft.UI.Xaml.Controls;
using Microsoft.UI.Xaml.Data;

namespace BMSMS.CustomControls
{
    11 references
    public sealed partial class VoltageCell : UserControl
    {
        3 references
        public int CellNumber { get; set; }

        private float _voltage;
        2 references
        public float Voltage
        {
            get
            {
                return _voltage;
            }

            set
            {
                _voltage = value;
                textBox.Text = value.ToString();
            }
        }

        1 reference
        public VoltageCell(int cellNumber, Binding b)
        {
            this.InitializeComponent();
            CellNumber = cellNumber;

            textBox.SetBinding(TextBlock.TextProperty, b);
        }
    }
}
```

CAN Message ID Enum:

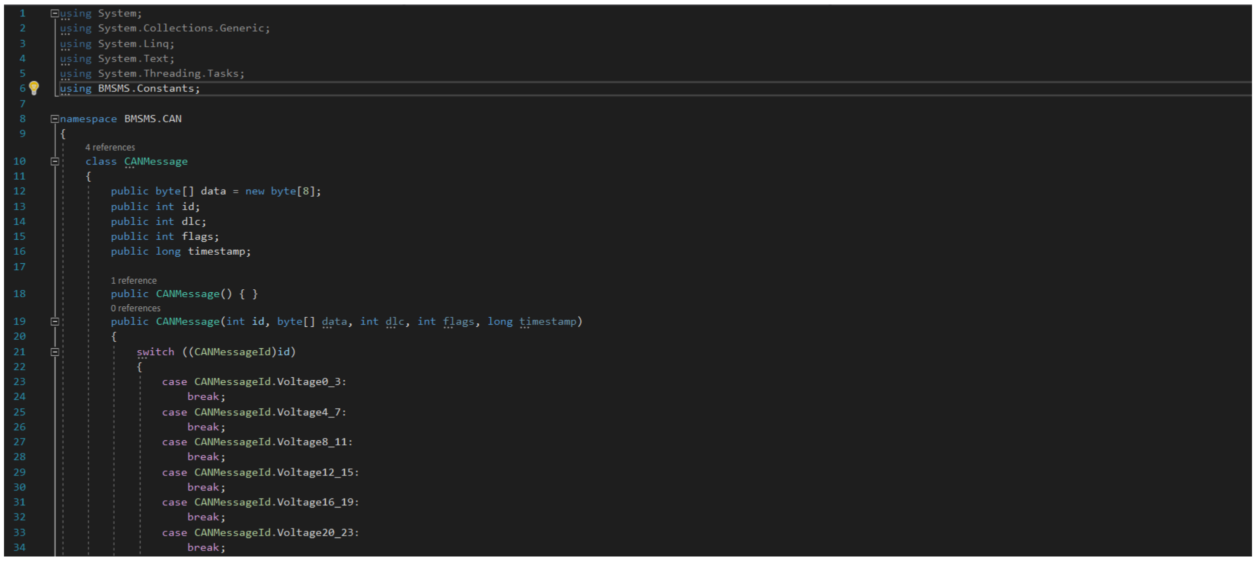


```
namespace BMSMS.Constants
{
    enum CANMessageId
    {
        //Voltages
        Voltage0_3 = 0x401,
        Voltage4_7 = 0x402,
        Voltage8_11 = 0x403,
        Voltage12_15 = 0x404,
        Voltage16_19 = 0x405,
        Voltage20_23 = 0x406,
        Voltage24_27 = 0x407,
        Voltage28_31 = 0x408,
        Voltage32_35 = 0x409,
        Voltage36_39 = 0x40A,
        Voltage40_43 = 0x40B,
        Voltage44_47 = 0x40C,
        Voltage48_51 = 0x40D,
        Voltage52_55 = 0x40E,
        Voltage56_59 = 0x40F,
        Voltage60_63 = 0x410,
        Voltage64_67 = 0x411,
        Voltage68_71 = 0x412,
        Voltage72_75 = 0x413,
        Voltage76_79 = 0x414,
        Voltage80_83 = 0x415,
        Voltage84_87 = 0x416,
        Voltage88_89 = 0x417,

        FuseBlown = 0x421,

        CellBalancing0_63 = 0x422,
        CellBalancing64_89 = 0x423,

        Temp0_3 = 0x424,
        Temp4_7 = 0x425,
```

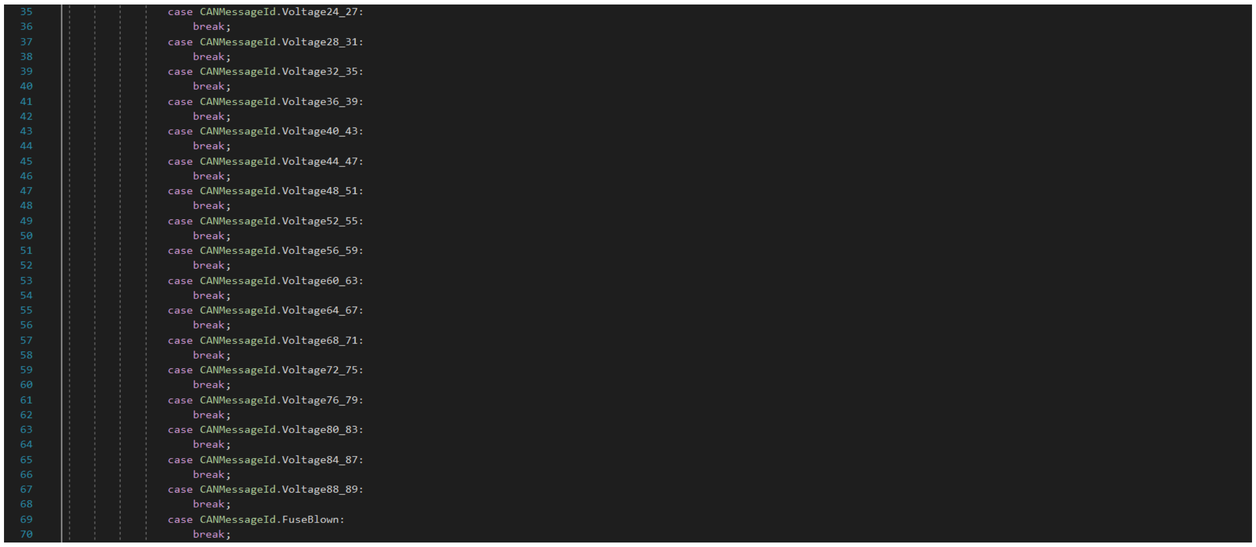


```
        Temp8_11 = 0x426,
        Temp12_15 = 0x427,
        Temp16_19 = 0x428,
        Temp20_23 = 0x429,
        Temp24_27 = 0x42A,
        Temp28_31 = 0x42B,
        Temp32_35 = 0x42C,
        Temp36_39 = 0x42D,
        Temp40_43 = 0x42E,
        Temp44 = 0x42F,

        AccumulatorData = 0x440
    }
}
```

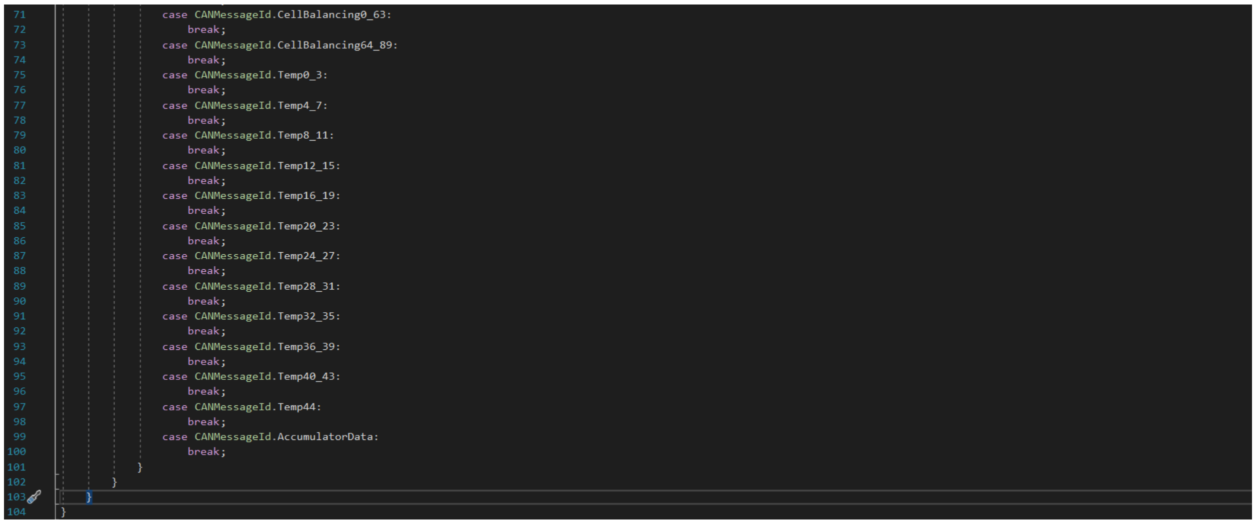
CAN Message Class:



```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;
using BMSMS.Constants;

namespace BMSMS.CAN
{
    class CANMessage
    {
        public byte[] data = new byte[8];
        public int id;
        public int dlc;
        public int flags;
        public long timestamp;

        public CANMessage() { }
        public CANMessage(int id, byte[] data, int dlc, int flags, long timestamp)
        {
            switch ((CANMessageId)id)
            {
                case CANMessageId.Voltage0_3:
                    break;
                case CANMessageId.Voltage4_7:
                    break;
                case CANMessageId.Voltage8_11:
                    break;
                case CANMessageId.Voltage12_15:
                    break;
                case CANMessageId.Voltage16_19:
                    break;
                case CANMessageId.Voltage20_23:
                    break;
```

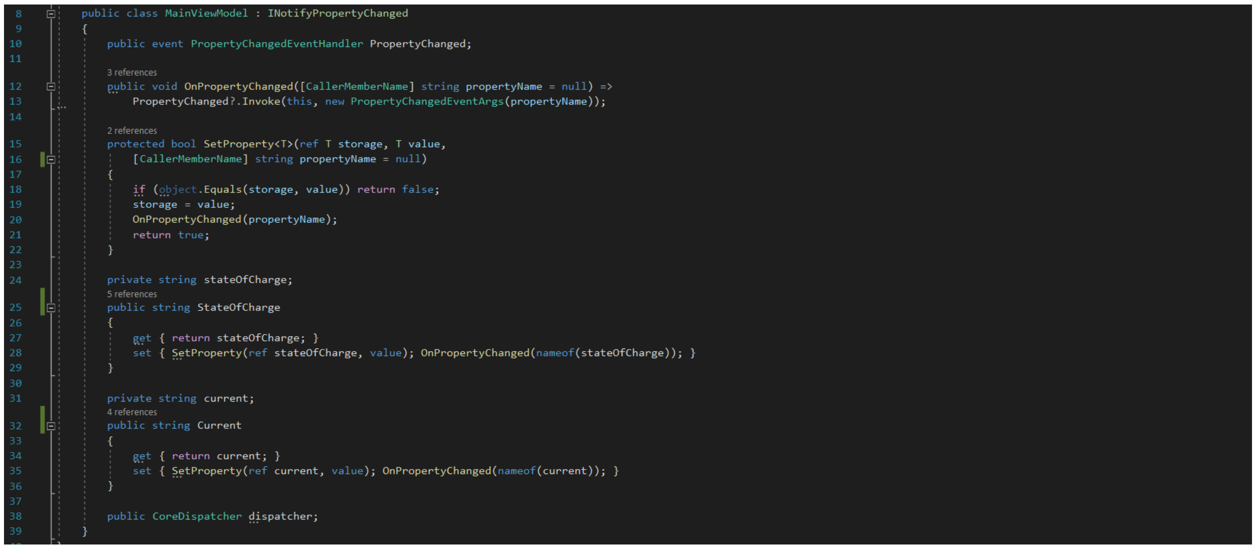


```
                case CANMessageId.Voltage24_27:
                    break;
                case CANMessageId.Voltage28_31:
                    break;
                case CANMessageId.Voltage32_35:
                    break;
                case CANMessageId.Voltage36_39:
                    break;
                case CANMessageId.Voltage40_43:
                    break;
                case CANMessageId.Voltage44_47:
                    break;
                case CANMessageId.Voltage48_51:
                    break;
                case CANMessageId.Voltage52_55:
                    break;
                case CANMessageId.Voltage56_59:
                    break;
                case CANMessageId.Voltage60_63:
                    break;
                case CANMessageId.Voltage64_67:
                    break;
                case CANMessageId.Voltage68_71:
                    break;
                case CANMessageId.Voltage72_75:
                    break;
                case CANMessageId.Voltage76_79:
                    break;
                case CANMessageId.Voltage80_83:
                    break;
                case CANMessageId.Voltage84_87:
                    break;
                case CANMessageId.Voltage88_89:
                    break;
                case CANMessageId.FuseBlown:
                    break;
```



```
                case CANMessageId.CellBalancing0_63:
                    break;
                case CANMessageId.CellBalancing64_89:
                    break;
                case CANMessageId.Temp0_3:
                    break;
                case CANMessageId.Temp4_7:
                    break;
                case CANMessageId.Temp8_11:
                    break;
                case CANMessageId.Temp12_15:
                    break;
                case CANMessageId.Temp16_19:
                    break;
                case CANMessageId.Temp20_23:
                    break;
                case CANMessageId.Temp24_27:
                    break;
                case CANMessageId.Temp28_31:
                    break;
                case CANMessageId.Temp32_35:
                    break;
                case CANMessageId.Temp36_39:
                    break;
                case CANMessageId.Temp40_43:
                    break;
                case CANMessageId.Temp44:
                    break;
                case CANMessageId.AccumulatorData:
                    break;
            }
        }
    }
}
```

Main View Model (where the data that is bound to the GUI is stored):



```
public class MainViewModel : INotifyPropertyChanged
{
    public event PropertyChangedEventHandler PropertyChanged;

    3 references
    public void OnPropertyChanged([CallerMemberName] string propertyName = null) =>
        PropertyChanged?.Invoke(this, new PropertyChangedEventArgs(propertyName));

    2 references
    protected bool SetProperty<T>(ref T storage, T value,
        [CallerMemberName] string propertyName = null)
    {
        if (object.Equals(storage, value)) return false;
        storage = value;
        OnPropertyChanged(propertyName);
        return true;
    }

    private string stateOfCharge;
    5 references
    public string StateOfCharge
    {
        get { return stateOfCharge; }
        set { SetProperty(ref stateOfCharge, value); OnPropertyChanged(nameof(stateOfCharge)); }
    }

    private string current;
    4 references
    public string Current
    {
        get { return current; }
        set { SetProperty(ref current, value); OnPropertyChanged(nameof(current)); }
    }

    public CoreDispatcher dispatcher;
}
```