Utah State University
DigitalCommons@USU

All Graduate Theses and Dissertations | Graduate Studies

8-2021

Control of Series Connected Battery Powered Modules

Joshua K. Wooten
Utah State University

Follow this and additional works at: https://digitalcommons.usu.edu/etd

Part of the Electrical and Computer Engineering Commons

Recommended Citation

Wooten, Joshua K., "Control of Series Connected Battery Powered Modules" (2021). *All Graduate Theses and Dissertations*. 8171.
https://digitalcommons.usu.edu/etd/8171

This Thesis is brought to you for free and open access by the Graduate Studies at DigitalCommons@USU. It has been accepted for inclusion in All Graduate Theses and Dissertations by an authorized administrator of DigitalCommons@USU. For more information, please contact digitalcommons@usu.edu.

CONTROL OF SERIES CONNECTED BATTERY POWERED MODULES

by

Joshua K. Wooten

A thesis submitted in partial fulfillment
of the requirements for the degree

of

MASTER'S THESIS

in

Electrical Engineering

Approved:

Regan Zane, Ph.D. Major Professor	Hongjie Wang, Ph.D. Committee Member
Yingying Zheng , Ph.D. Committee Member	D. Richard Cutler, Ph.D. Interim Vice Provost of Graduate Studies

UTAH STATE UNIVERSITY
Logan, Utah

2021

Copyright © Joshua K. Wooten 2021

All Rights Reserved

ABSTRACT

Control of Series Connected Battery Powered Modules

by

Joshua K. Wooten, Master's Thesis

Utah State University, 2021

Major Professor: Regan Zane, Ph.D.
Department: Electrical and Computer Engineering

Second life battery implementation is a method to reuse battery cells and help the environment. Voltage of a single battery cell is much lower than the total output voltage for high-voltage (HV) and high-powered systems. Due to this, many cells need to be connected in series to achieve higher voltage. As a result of the unavoidable mismatch in cell charge in the second life batteries, there is different amounts of degradation that happens within each cell, causing the entire battery pack to fail once the lowest performing battery cell fails. For this reason, systems that employ many cells connected in series utilize a battery management system (BMS). This BMS sends commands and employs the use of a direct current (DC/DC) converter to achieve the desired output voltage. Designing this system to be modular allows for the scalability eventually needed in the future with larger systems. This thesis looks to explain the control strategy for implementing active cell balancing among multiple cells connected in series. This is done by regulating the output voltage and utilizing voltage sharing among the multiple modules.

(96 pages)

PUBLIC ABSTRACT

Control of Series Connected Battery Powered Modules

Joshua K. Wooten

Batteries are a very common type of power source used for all sorts of applications. However, these batteries do not last forever. The purpose of this thesis is to explain and implement a strategy which allows batteries to have a second life; to be able to be recharged and re-used for another set period of time once they no longer meet a system's requirements. The preferable way to do this is to make each cell have the same state of charge (SOC) which will lead to a longer lasting battery pack. These batteries will be hooked up to a battery powered module (BPM) for charging, and these BPMs will need to be controlled by a battery management system (BMS). This BMS, also known as string controller board, sends commands to the several BPMs connected. This thesis goes over the strategy implemented to achieve this objective, with the result focusing on the regulation of output voltage and cell SOC.

To my family.

ACKNOWLEDGMENTS

The work presented in this thesis would not be possible without the amazing support from many people in my life. I want to express my thanks to my major professor, Dr. Regan Zane. Without your many hours invested in my education and learning, I would not be where I am today academically. Thank for your time, dedication, and support to me and this thesis. I also want to thank you for the opportunity to work at the Utah State Power Electronics Lab (UPEL). It has been an amazing learning experience, and something that I am extremely grateful for. I have gotten a lot of hands-on experience from this opportunity and have met a lot of great engineers and individuals along the way.

I would like to thank my committee members: Dr. Hongjie Wang, and Dr. Yingying Zheng for their comments and support throughout the thesis process.

I would like to thank my fellow colleagues at UPEL Dr. Mohamed Kamel, Dr. Rohail Hassan, and Brooks Maughan. There were countless hours spent discussing challenges and tactics to complete this thesis. Your guidance helped me tremendously.

Lastly, I would like to thank my family and especially my wife, Shalee Wooten, for their unwavering support of me at home throughout my education.

Joshua Kade Wooten

CONTENTS

Page

ABSTRACT . . . iii

PUBLIC ABSTRACT . . . iv

ACKNOWLEDGMENTS . . . vi

LIST OF TABLES . . . ix

LIST OF FIGURES . . . x

ACRONYMS . . . xii

1 INTRODUCTION . . . 1
- 1.1 Objectives . . . 2
- 1.2 Overview . . . 2

2 LIT REVIEW AND BACKGROUND . . . 4
- 2.1 Control of DC/DC Converters . . . 4
- 2.2 Control of Series DC/DC converters . . . 7
  - 2.2.1 Power Management With A Voltage Map . . . 8
  - 2.2.2 Current Regulation Approach . . . 10
  - 2.2.3 Droop control . . . 10

3 METHODS AND MATERIALS . . . 13
- 3.1 Background . . . 14
- 3.2 PCB Design . . . 15
  - 3.2.1 PCB Fabrication . . . 17
- 3.3 CAN Communication and ADC code implementation . . . 19
  - 3.3.1 ADC implementation . . . 24
- 3.4 Control of series connected BPMs . . . 24
  - 3.4.1 Droop control and delta current command . . . 26

4 RESULTS AND ANALYSIS . . . 29
- 4.1 String controller . . . 29
- 4.2 CAN communication and ADC sensing . . . 30
- 4.3 System Control . . . 32
  - 4.3.1 Single module control . . . 32
  - 4.3.2 Series connected module control . . . 35

5 CONCLUSION AND FUTURE WORK . . . 42
- 5.1 Future work . . . 42

REFERENCES . . . 44

APPENDICES 46
A String controller code 47

LIST OF TABLES

Table Page

3.1 PCB costs. 18

3.2 CAN message structure 1. 22

3.3 CAN message structure 2. 23

3.4 ADC Linearization. 25

4.1 ADC test. 31

4.2 Droop results 38

4.3 Series control droop results 39

LIST OF FIGURES

Figure Page

2.1 Module Controller Topology [1] 5
2.2 Control approach [1] 7
2.3 Voltage map [2] 9
2.4 Current Control [2] 11
2.5 Current Control with droop [2] 12
3.1 Full power system. 14
3.2 Schematic. 16
3.3 PCB in Altium. 17
3.4 Populated PCB. 19
3.5 CAN bus [3]. 20
3.6 CAN message breakdown [4] 20
3.7 CAN communication. 21
3.8 IQ wrapping. 24
3.9 ADC Linearization plot. 26
4.1 Power test. 30
4.2 Continuity checks. 31
4.3 ADC circuit tests 32
4.4 CAN test setup. 33
4.5 CAN test results. 33
4.6 Initial module test 34
4.7 Module Voltages. 35

4.8 Droop test set up. 36

4.9 Droop results. 37

4.10 Two cycles of cell balancing. 39

4.11 Two module set up. 40

4.12 Series control. 41

4.13 Series control with droop sweep. 41

ACRONYMS

UPEL	utah state university power electronics lab
DC	direct current
SOC	state of charge
SOH	state of health
BPM	battery powered module
BMS	battery management system
EV	electric vehicle
kW	kilowatt
POC	proof of concept
ISR	interrupt service routine
ZVS	zero voltage switched
PWM	pulse width modulation
PnP	plug and play
CAN	control area network
UTP	unshielded twisted pair
PCB	printed circuit board
PWM	pulse width modulation
HV	high voltage
LED	light emitting diode
IC	integrated circuit
FPGA	field programmable gate array
PC	personal computer

CHAPTER 1

INTRODUCTION

Power systems that have bidirectional power flow require a certain control technique to achieve direct current (DC) bus voltage regulation and output voltage sharing among the power converters. The control strategy for this thesis looks to achieve cell balancing and distribute power equally among each modules' converter. This will be done according to each modules' state of charge (SOC) and state of health (SOH). [1] Balancing lithium-ion battery cells is a crucial factor in power electronics. It is frequently discussed when multiple cells are used in a battery pack. Balancing these cells' SOC allows for a longer lifespan for the batteries. This increases efficiency and reliability for these batteries. Having many of these battery cell interfacing modules connected in series presents challenges to control that will be covered in this thesis.

This thesis aims to create a scalable, robust, and modular system to realize large lithium-ion battery packs with active cell balancing. More specifically, the project strives to implement this for second life batteries. This will allow for batteries that wear down and don't meet their initial requirements to have a second life. Second life opportunities for batteries will help the world in many ways. First, it is Eco-friendly due to the fact that not as many batteries will need to be made each year. Each battery saved and reused is one less battery that needs to be manufactured. Second, this will increase technology for electric batteries and incentive these over the more harmful for the environment lithium batteries. Third, this advancement in balancing second life batteries will also help with taking battery cells from different companies, manufacturers, and systems and combining them into one battery pack for another system.

This thesis looks to analyze and develop the most effective way to use a microcontroller to communicate information back and forth between itself and another microcontroller and control what happens. This controller (on the BPM) needs to know each cell's respective

available power and SOC. [4] This is done by measuring each cell's current and voltage then communicating the information through a controller area network (CAN). Another key objective of the string controller is to control the droop current. This will be directly dependent on the power and SOC available in each cell.

1.1 Objectives

The main objective of this thesis is to show optimal and functional control of series connected modules which are also connected to battery cells.

To achieve this major objective, this thesis will prove the completion of the following three objectives:

- Design, develop, and build a hardware controller that is able to perform the functions necessary as stated in the beginning of the introduction 1.
- Develop software to realize sensing the bus voltage of the series connected battery power modules (BPM)s and communicate this, along with other information through CAN.
- Develop control algorithms to achieve desired SOC and cell balancing among cells, as well as droop control among the modules.

These objectives are important to accomplish so that the final system has active cell balancing which will lead to batteries having a second life with better efficiency.

1.2 Overview

The remaining chapters of this paper discusses details pertaining to the project. Chapter 2 reviews the current work and research that has already been explored and published by credible researchers. It discusses converter topology and control schemes previously explored by other engineers. Chapter 3 discusses the approaches and techniques used to achieve this thesis' objectives. Chapter 4 discusses and shows the results obtained from the system as well as how the tests are performed It also proves that the objectives were

met. Chapter 5 summarizes and reiterates the impact and importance of this project on the world today.

CHAPTER 2
LIT REVIEW AND BACKGROUND

This chapter talks about the work done by other great students, professors, and specialists in this particular field. This work helps pave a way for the accomplishment of the objectives previously stated in Chapter 1. A battery powered module (BPM) is a processing DC/DC converter that integrates the battery management system (BMS) [2]. The purpose for using a DC/DC converter and BMS is to achieve cell balancing with battery cells that are connected to the modules. The idea of using DC/DC converters to perform active battery cell balancing is discussed thoroughly in [5–9]. While the actual full approach is complex, the main idea is to charge and discharge each cell at different rates based on the cells' SOC. After enough time and cycles, each cell will be at the same SOC which accomplishes the goal of active cell balancing.

Second life battery applications have been looked into by many great researchers. Lithium-ion batteries typically preserve 70%-80% of their initial capacity when retired from their electric vehicle (EV) use [10]. While these batteries are no longer capable for their initial use, they can still provide energy storage services in less-demanding applications. Balancing the uniformity of capacity among the battery cells is crucial to the success of second life applications. [10] talks in detail about the effects and results of lithium-ion batteries SOH and ageing history over second life performances.

2.1 Control of DC/DC Converters

A very similar topology has already been looked into thoroughly [1]. The difference being that the experiments and data collected are only done to a converter with three primary ports instead of four. The module controller topology is illustrated in Fig. 2.1. Along with this, the beginning stages of development for this type of topology using multiple input ports to a DC/DC converter are discussed in [11, 12]. Here the idea of using a two-

phase interleaved operation using pulse width modulation (PWM) is discussed. In these papers, the idea of using differential dual-active bridge converter is discussed as well. More topology options are introduced in [13, 14]. These papers focus on improvements that can be made to achieve active cell balancing cheaper, more efficient, and with less components.

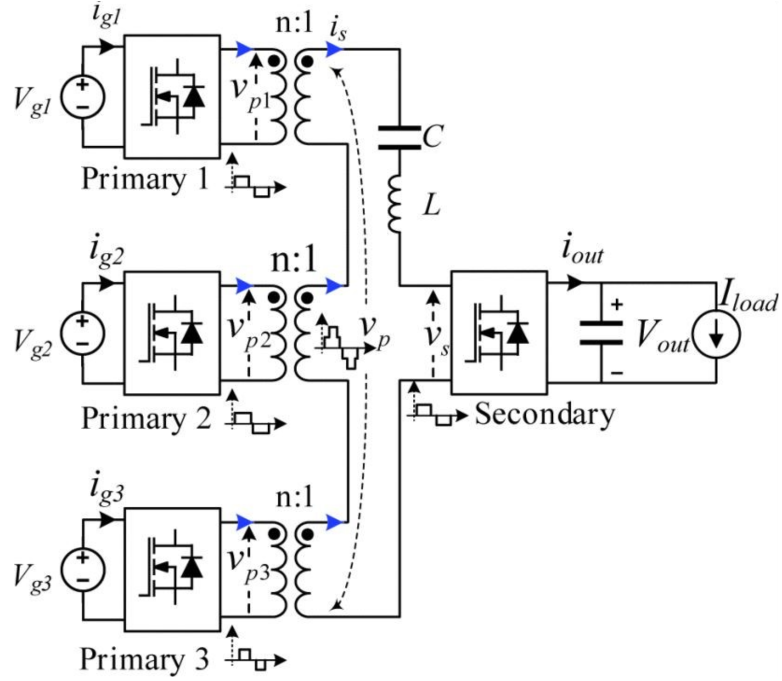


Fig. 2.1: Module Controller Topology [1]

Even though the converter topology is not the same, most of the same data and findings can be applied to the work of this thesis. One main finding that this paper discusses is the idea of switching from multiple secondary ports (one for each primary port) to just one secondary port. While the first technique does technically work, it leads to a much larger power loss. This is because each secondary winding would need to make connections between adjacent terminals, causing the winding length and overall resistance for the transformer to increase.

A complete derivation for this topology seen in Fig. 2.1 can be found in [1]. This thesis focuses on the concepts discovered and main control methods that will be useful for this thesis. The end equations are referenced below:

$$V_{p,k} = 4/\pi * sin(\alpha_k/2) \tag{2.1}$$

Where V_{pk} is the amplitude of the fundamental phasor for that specific primary input port. k is referenced as the specific port number.

$$v_{p,k} = V_{p,k}\,(cos\phi_k + jsin\phi_k) \tag{2.2}$$

$$I_{g,k} = 1/2X_s * [V_s sin(\phi_k) + V_{p1} sin(\phi_1 - \phi_k) + V_{p2} sin(\phi_2 - \phi_k) + ... + V_{pm} sin(\phi_m - \phi_k] \tag{2.3}$$

Where X_s is the impedance of the resonant tank and is given by the equation 2.4

$$X_s = w_s L - 1/w - nC \tag{2.4}$$

$$P_{out} = V_{out}/2X_s * [V_{p1} sin\phi_1 + V_{p2} sin\phi_2 + ... + V_{pm} sin\phi_m] \tag{2.5}$$

$$I_{out} = 1/2X_s * [V_{p1} sin\phi_1 + V_{p2} sin\phi_2 + ... + V_{pm} sin\phi_m] \tag{2.6}$$

For equations 2.3, 2.5, and 2.6 m is represented as the total number of input ports.

The primary port has two control parameters, which are the duty cycle and the phase-shift. The power delivered to the secondary depends on both values. [1]. The secondary port is kept constant and not used as a control parameter. Since each primary port has a different duty cycle, but all share a rising edge, all the primary bridges are zero voltage switched (ZVS). This means that they are all turned on at the same time. Using this approach allows for the control of the relative duty cycles, which in turn controls the relative loading between cells by applying a direct relationship to the current. It is important to know each port's duty cycle α to calculate the current I_{gk}.

This approach is accomplished using a weighted system control loop. A figure of this can be seen in Fig. 2.2. The first step is to determine which port has the highest duty cycle. This is given the weight of 1 and given the highest input current that is also used as the base current i_{base}. The weight reference is then calculated for each other port based on their respective duty cycle and used in a control loop to achieve the desired current. This achieves the goal of cell balancing.

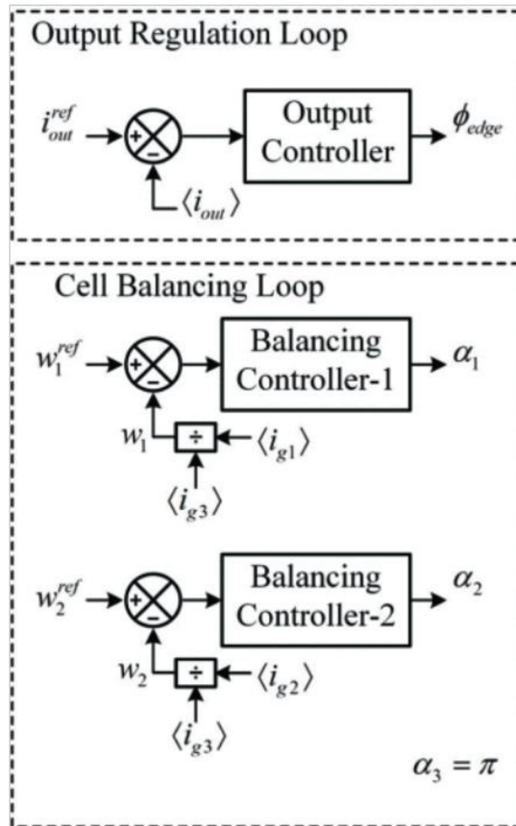


Fig. 2.2: Control approach [1]

2.2 Control of Series DC/DC converters

There are two ways that a higher output voltage for the system can be achieved. The turns ratio for the converter can be increased on the secondary side, or more converters can be placed in series. The series connected converters option is chosen because it allows

for a higher efficiency system. While increasing the turns ratio does technically achieve the same goal, the efficiency drastically drops for this case. Section 2.1 has discussed the idea of BPM as a building block for larger battery packs. This section focuses on placing another BPM in series and the control approach behind this. The first step for control of series DC/DC converters is to regulate each DC/DC modules' voltage. This can be done for each individual module by comparing its output voltage to a desired reference. The overall output voltage of the system can then be increased by adding more modules in series. [2]. In order to allow for more modular designs, which allows for a larger DC bus voltage range and Plug-and-Play(PnP) operation, it is necessary to integrate SOC control, current control, and droop control. These topics will be covered more in depth in section 2.2.1, section 2.2.2, and section 2.2.3.

The topic of control for balancing battery cells has been the focus of many electric vehicle researchers for quite some time. [15, 16] discusses the basics of a BMS and the techniques used for battery balancing. [17] presents an energy sharing SOC balancing control scheme. [18] discusses a control technique for cell balancing with a two stage converter. [6] discusses a module-integrated distributed battery energy storage and management system without the need for additional battery equalizers. This is important because it allows multiple cells to be controlled through a single DC/DC converter. [19–22] present additional control techniques for various converter topologies.

2.2.1 Power Management With A Voltage Map

The idea behind using a voltage map is to be able to see a cell's voltage and figure out the rate at which that cell needs to be charged or discharged. Fig. 2.3 shows an example of a voltage map that is divided into both a charging and discharging section.

To find the value K_{chg}, it is necessary to find the linear slope. This can be seen in equation 2.7

$$K_{chg} = (SOC_{max} - SOC_{min})/(V_4 - V_3) \tag{2.7}$$

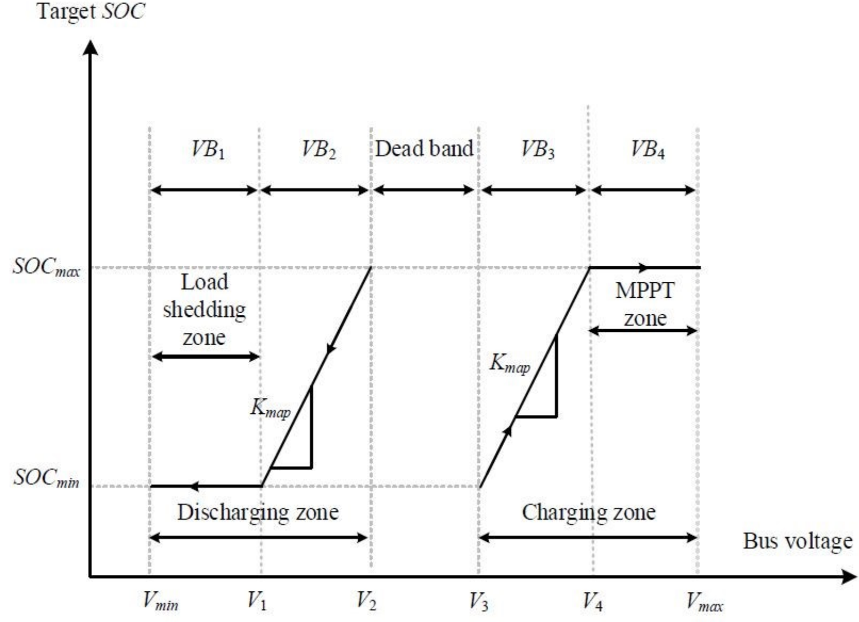


Fig. 2.3: Voltage map [2]

K_{chg} value allows for SOC_{chg} to be calculated. This makes it so that SOC_{avg} can be regulated. An example of this is demonstrated in 2.4.

$$SOC_{chg} = K_{chg}(V_{bus} - V_3) + SOC_{min} \tag{2.8}$$

K_{dis} follows the same approach as K_{chg}, and in the end it can be seen that they are equivalent. This can be seen in equation 2.11.

$$K_{dis} = (SOC_{max} - SOC_{min})/(V_2 - V_1) \tag{2.9}$$

$$SOC_{dis} = K_{dis}(V_{bus} - V_1) + SOC_{min} \tag{2.10}$$

K_{map} is the same for discharging and charging.

$$K_{map} = K_{dis} = K_{chg} \tag{2.11}$$

$$SOC_{ref} = K_{map}V_{bus} \tag{2.12}$$

2.2.2 Current Regulation Approach

The values of each cells' SOC is sent over CAN and used by the string controller to find the SOC_{avg}. The SOC_{avg} is then regulated to SOC_{target} which is determined based on the bus voltage that is sensed and the corresponding k_{map} value that has been discussed in section 2.2.1. The string controller is then able to calculate i_{all}.

$$i_{all} = Gscoc(SOC_{avg} - K_{map}v_{bus}) \tag{2.13}$$

Due to the fact that $SOC_{ref} = K_{map}V_{bus}$ as seen in 2.12, plugging this into 2.13 gives 2.14

$$i_{all} = Gscoc(SOC_{avg} - SOC_{ref}) \tag{2.14}$$

This i_{all} value is then transmitted over CAN to each module controller. This will be used by each corresponding module controller to determine the rate the cells are charged/discharged. Essentially the current reference is used to regulate the pack average SOC. The control structure can be seen in Fig. 2.4. i_{all} is then used as one of the main inputs within the BPM.

2.2.3 Droop control

In order to help regulate the output voltage and make sure that the bus voltage is shared equally with all modules, a droop loop is added. Fig. 2.4 is then modified with the addition of the droop loop. This addition can be seen in Fig. 2.5. Droop control is enabled to bring this particular module output voltage closer to the desired reference of v_{avg} which is then passed through a gain G_d. See [2] for finding G_d. This droop loop will add or subtract a small amount of current depending on the relation between the output

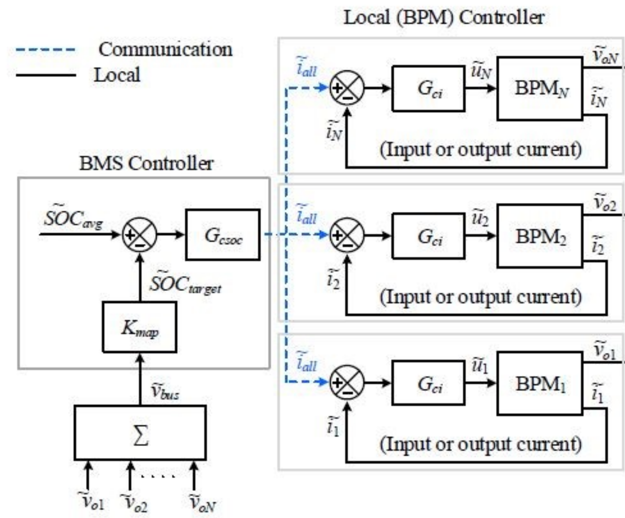


Fig. 2.4: Current Control [2]

voltage and v_{avg}. This current is called the delta current command and can be seen in the figure as i_{dN}, where N represents which number module it is.

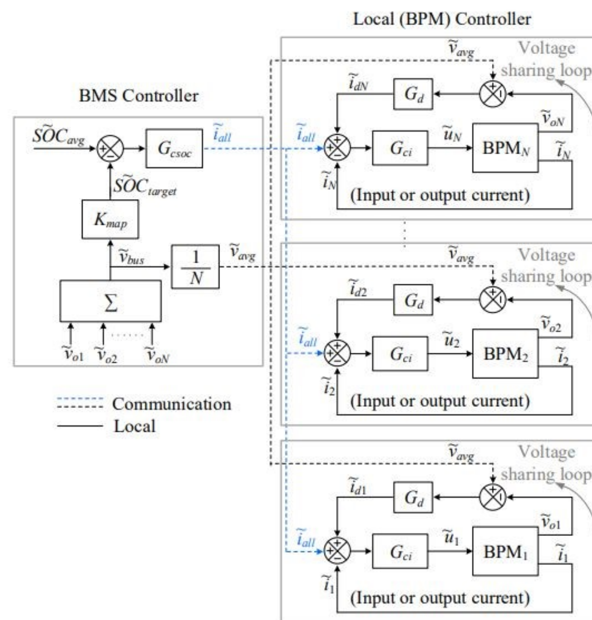


Fig. 2.5: Current Control with droop [2]

CHAPTER 3
METHODS AND MATERIALS

This chapter focuses on the system setup, techniques used, and methods performed to obtain the project objectives as stated in Chapter 1. The project takes the techniques used for implementing series BPM control as addressed in Chapter 2 and applies it to this project's specific system. It also makes some adjustments for what the system needs. This is accomplished by developing a controller to communicate commands back and forth between the battery powered modules (BPM)s and site controller. The site controller and other parts of the system will be explained in section 3.1 as it gives an overview of the entire system.

This thesis proposes an isolated multi-port converter that has eight primary ports with the use of H-bridges to one secondary ports. The purpose for this topology is to achieve active cell balancing and allow four, 4V cells to reach 32V output on the secondary side, while not needing a DC/DC converter for each cell. The advantages for this topology are as stated:

- Allows use of the same ground reference for all cells, eliminating the need for isolated communications between active balancing converters.
- Reduced component count due to the fact that there is only one secondary port.
- The magnetics are efficient since the converters are physically close, the secondary winding can be wound continuously through all the cores reducing the winding length.
- Elimination of contact resistance since multiple cells are able to connect to one DC/DC converter.

[1]

The rest of the sections in this chapter break down the steps needed to accomplish these tasks. They are organized as follows: Section 3.1 gives an overview of the system and

discusses the options for the controller implementation. Section 3.2 discusses the process of designing and developing the PCB for the controller. Section 3.3 talks about implementing the communication protocol and sensing done through the ADC. Section 3.4 discusses the control techniques to send the correct current command to the multiple BPMs.

3.1 Background

The full 100 kilo-watt (kW) power system layout can be seen in figure 3.1. The full power system entails a site controller that will control all of the string controllers. Every string controller controls and communicates with six module controllers. Each module controller has sixteen battery cells that it is connected to and senses. These cells have been previously used and provided from Nissan and Tesla. This thesis focuses on one branch of string controller that is connected to two module controllers. Two module controllers interfacing with a string controller is what is needed to show series connection control. Each module controller for this thesis will be connected to four battery cells. This is what is called a proof of concept (POC) system which focuses on demonstrating capability. Chapter 1 goes over the full objectives for this thesis.

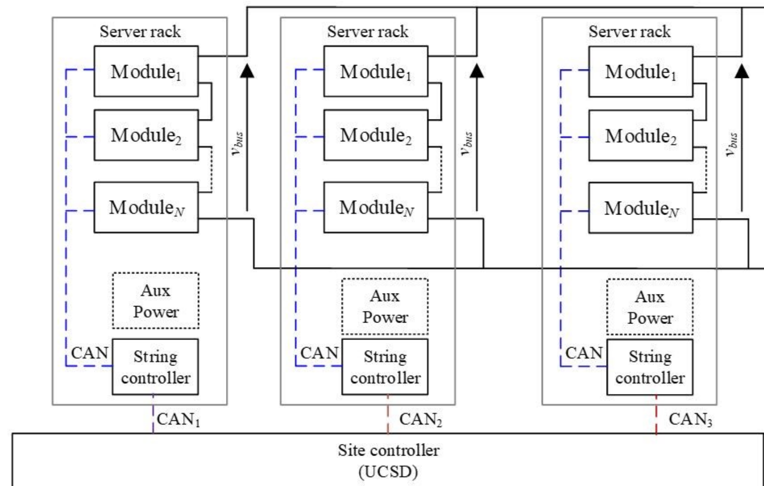


Fig. 3.1: Full power system.

There are several ways to approach the challenge of control. The first choice is to

have everything done in the string controller. This entails having every control loop, all the calculations, and all the sensing done on one device. The string controller then sends out any command or information to the module controller. This allows for every control loop to be on the same frequency and everything to be rather quick. The downside to this approach is the fact that later, the system would not be able to be scaled upward. There would be issues when adding more BPMs in series or additional strings in parallel. The flip side of this strategy is to do everything in the BPM and eliminate the string controller all together. This eliminates confusion on which controller will perform each task, connection issues, and causes there to be less noise among the communication lines. The reason this approach is not ideal is because it is very difficult for the module controllers to be able to coordinate, especially during start up. It is very important to be able to ramp up the duty cycle and output voltage at the same time.

The approach chosen lies somewhere in the middle. Some aspects are controlled in the BPM and some are controlled in the string controller. This happy middle ground allows for some of the pros while avoiding the worst cons. It is possible that in the future an alternative solution proves to be more effective.

3.2 PCB Design

The string microcontroller requires a PCB with additional circuits to fulfil its functions. The first step in designing a PCB for a controller is to think of all the purposes that are needed for the board and lay it all out in a schematic. For the case of this project, the string controller needs several things. First, a microcontroller that allows code to process information and send out commands to the module controller. Second, transformers that step down the voltage from the 12V auxiliary input to 5V and 3.3V. This was used to power other devices on the PCB, with one of those being the microcontroller. Third, isolated communication area network (CAN) transceivers for the CAN communication that is discussed further in section 3.3. Fourth, a circuit used to implement ADC sensing of the series connected bus voltage. This is discussed more in depth in section 3.3. Lastly it is necessary to include additional smaller items such as a debug port, crystal oscillator that is

used for the microcontroller clock, and additional headers that are used to connect hardware to the board. It is important to show how all of this is connected to the microcontroller and take note of which pins are to be used [1]. The finished schematic for this project's PCB is illustrated in Fig.3.2. Altium has a great resource for getting started with schematics. [23]

Fig. 3.2: Schematic.

The next step with the PCB process is to import the schematic into a PCB file. This places all the components and a highlighted net showing which components are connected and an outline of a board with dimensions specified by the user. The next step is to specify the number of layers needed for the board and decide which components will go on the top layer or the bottom layer. Component placement takes some time but becomes easier with time and experience. A good practice is to review component schematics which may have example layouts shown. Another tip is to start with components that need to be close together and even consider placing those components right on top of each other, just on opposite sides of the board. Following this, copper traces between components are added.

It is important to verify that the traces are large enough if power is to be flowing through them. The final step is to run the debugger and check that there are no issues that will be seen by the manufacture and to add a keep out layer for them to see. This project has ensured the following of these steps to avoid major issues with the completed board. The finished PCB layout is illustrated in Fig. 3.3.

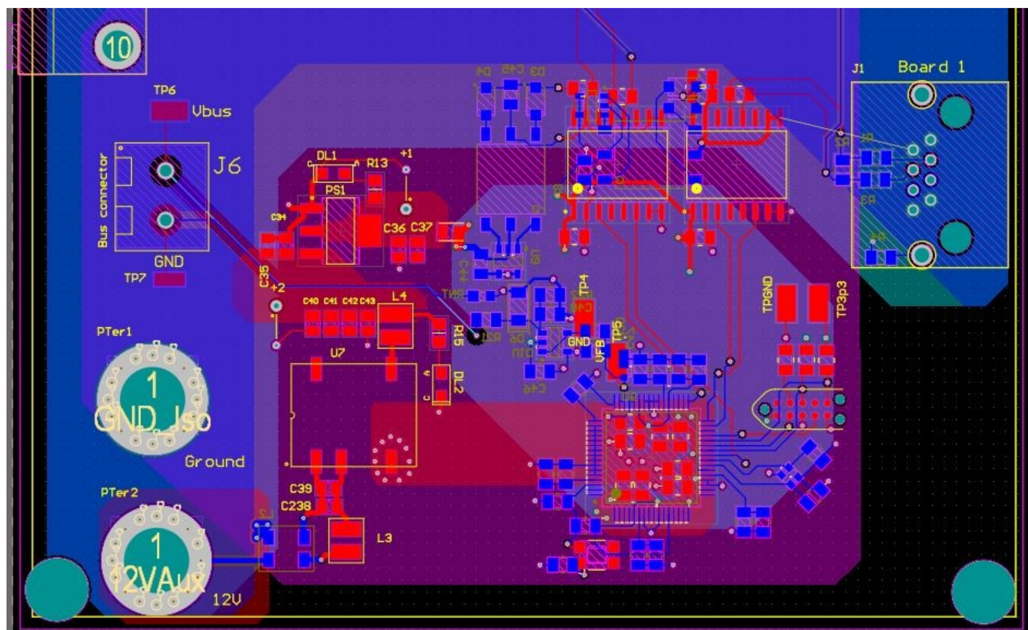


Fig. 3.3: PCB in Altium.

3.2.1 PCB Fabrication

Upon completion of the PCB layout, the next step is to order the fabrication of the board and the components that would then be placed onto it such as resistors, capacitors, and integrated circuits (IC). There are three options that this task can be carried out. They

Table 3.1: PCB costs.

PCB order method	Cost
PCB with populated components	8,526.6
PCB with stencil	4,354.08
PCB	4,154.08

are as follows:

- The PCB is ordered along with the bill of materials needed for the components on the board. The manufacturer is responsible for the board's population for an extra cost.
- A stencil is ordered along with the PCB that has cutouts of all the components. This will allow for the engineer to use solder paste to place all components quicker than all by hand.
- Nothing additional to the PCB is ordered, and the components are all soldered by hand by the engineer ordering it.

These three options all come with different price points and a different amount of work needed by the engineer ordering it to get the PCB completed. An analysis for the full power system PCB showing the different costs for the three different price points is shown in table 3.1. This analysis consists of 10 PCBs instead of one.

The second option, which includes the PCB and stencil, is chosen. This is another example of how this project lies in a happy middle ground. The company can save on costs while also assisting the engineer with time by using the stencil to solder both sides of the board. There are cases where each option is the most preferred, depending on the status of the project and desires of those funding it.

An additional thing to note is that due to the second method being chosen, the components need to be ordered around the same time as the PCB. This allows for everything to arrive at about the same time. Using the stencil makes it so the components can be placed gently on the board and then once the entire side of components have been place,

re-flowed in the oven. This heats the paste and causes the paste to behave like actual solder to form an electrical connection between the board and components. Using the stencil allows for much cleaner looking connections then what one would be able to do by hand. The populated (PCB) is illustrated in Fig. 3.4.

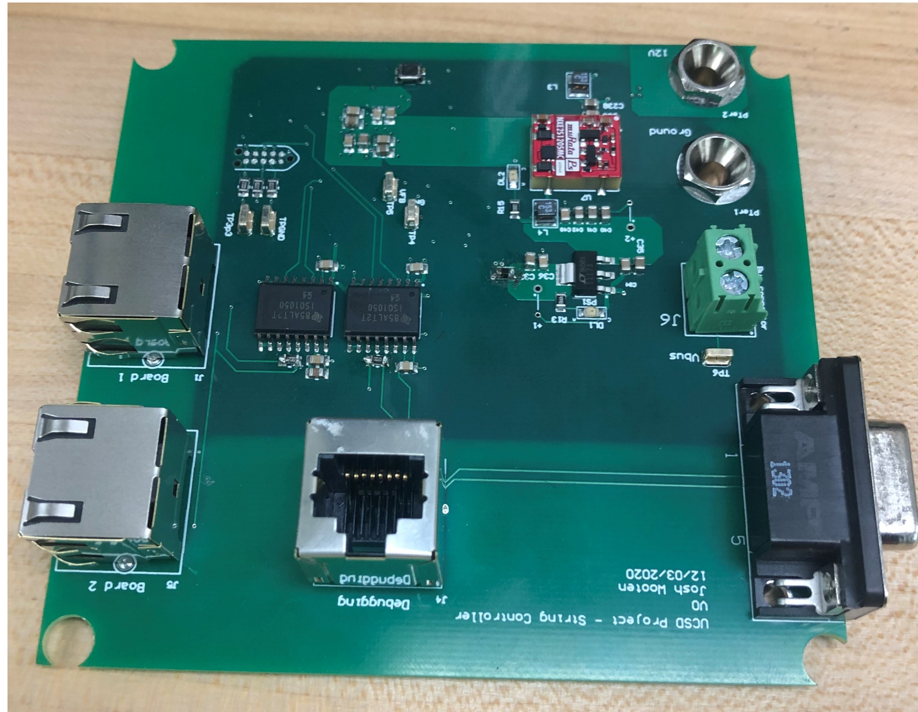


Fig. 3.4: Populated PCB.

3.3 CAN Communication and ADC code implementation

CAN is a communication protocol that allows microcontrollers and devices to communicate with each other without a host computer. For this project, CAN allows the string controller the ability to talk with multiple module microcontrollers and send commands back and forth across the CAN bus. The CAN bus is illustrated in Fig. 3.5. This is done by connecting these module microcontrollers to the same CAN bus line using an unshielded twisted pair (UTC).

The CAN bus line allows unique messages that are meant for one specific device to be

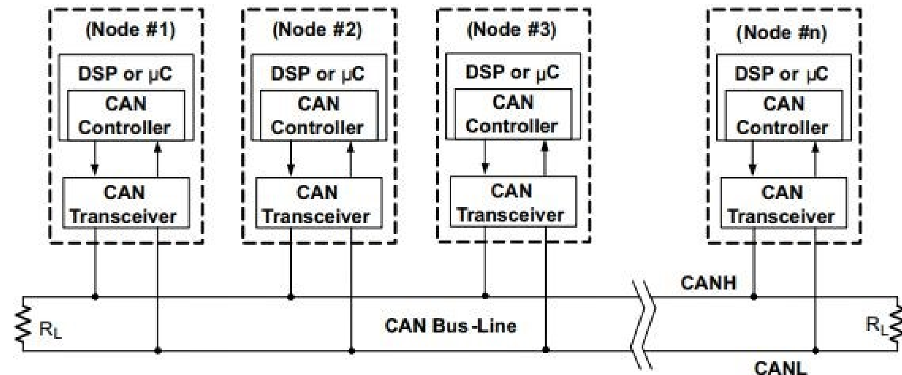


Fig. 3.5: CAN bus [3].

received by that device. This is done by using an 11-bit unique identifier. This identifier also represents the message priority of the device. The lower the number, the higher the priority and quicker that that message will be sent/received. [3] A full diagram of a CAN message can be seen in Fig. 3.6.

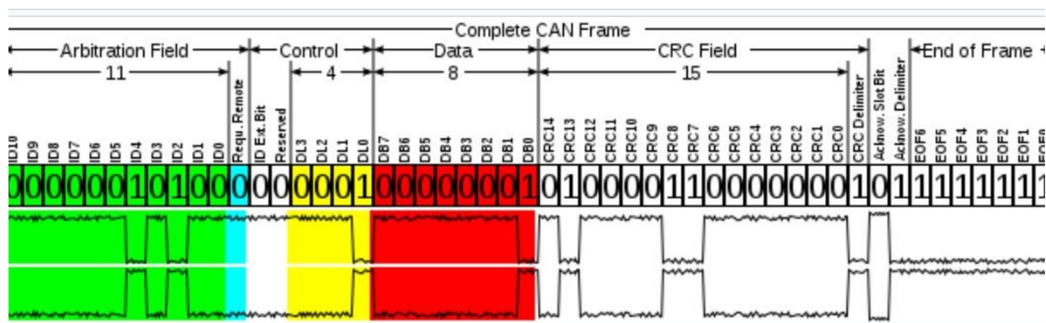


Fig. 3.6: CAN message breakdown [4]

To set this communication up in code, two main steps are required. The first step is to initialize the CAN bus. This includes setting up the clock source, choosing desired settings for the CAN communication and setting up mailboxes which are all the parts needed in a CAN message. This again can be seen in Fig. 3.6. The second step is to enable and configure the Interrupt Service Routine (ISR). This will allow for certain code to be executed when the microcontroller detects data meant for it on the CAN bus. Within the ISR it stores

the data it receives into local variables and performs any other task that is chosen by the programmer. This project sends important values over the CAN bus that are needed in within the module controller and site controller. The communication needed between the string controller and the DC/DC converter can be seen in Fig. 3.7. Those communication lines are marked with a red check mark. This diagram also shows the larger system. See section 3.1

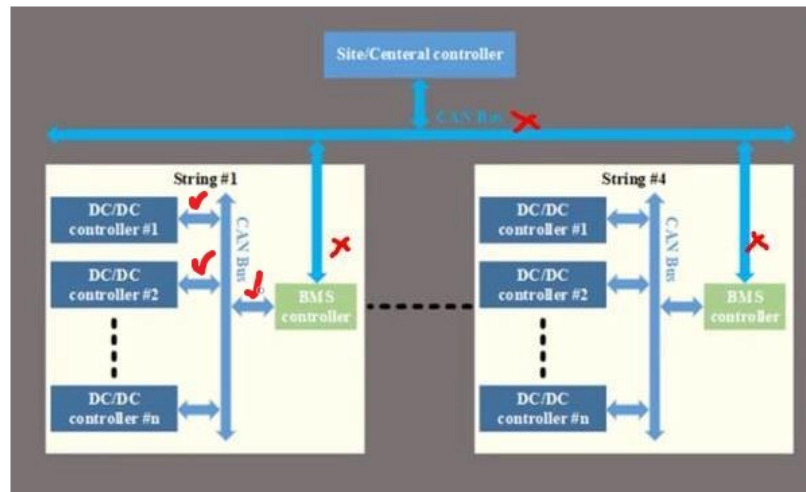


Fig. 3.7: CAN communication.

There are a few different CAN messages sent to the module controller, string controller, and the personal computer (PC) console. The various messages sent to the PC console can be seen below in table 3.2.

The same values for any number of module controllers are also sent to the PC console. The CAN message structure for string controller and module controller can be seen in table 3.3

As seen in table 3.2 and table 3.3, the values for a single variable are sent in two data positions of the array. As the message is received, the microcontroller unpacks the data using IQ wrapping. IQ wrapping is used to avoid losing precision with data sent. Since the data is sent over CAN as integers 8 bits at a time, IQ wrapping is a way around this and allows the user to send decimal numbers. In this project's case it is very crucial that this

Table 3.2: CAN message structure 1.

Destination	Array number	Variable
MATLAB	0	SOC value for cell 1
	1	SOC value decimal portion for cell 1
	2	SOC value for cell 2
	3	SOC value decimal portion for cell 2
	4	SOC value for cell 3
	5	SOC value decimal portion for cell 3
	6	SOC value for cell 4
	7	SOC value decimal portion for cell 4
MATLAB	0	Voltage value for cell 1
	1	Voltage value decimal portion for cell 1
	2	Voltage value for cell 2
	3	Voltage value decimal portion for cell 2
	4	Voltage value for cell 3
	5	Voltage value decimal portion for cell 3
	6	Voltage value for cell 4
	7	Voltage value decimal portion for cell 4
MATLAB	0	Current value for cell 1
	1	Current value decimal portion for cell 1
	2	Current value for cell 2
	3	Current value decimal portion for cell 2
	4	Current value for cell 3
	5	Current value decimal portion for cell 3
	6	Current value for cell 4
	7	Current value decimal portion for cell 4

is implemented. For example, it is undesirable to send the number 4 as the voltage for a particular cell, when it is actually 4.01687.

IQ wrapping works by placing the value in a variable type "$IQ12$". The number after IQ represents how many bits make up the data after the decimal. The rest of the bits make up the data before the decimal. For this case, the remaining bits would be 4, since 16 bits of data are being sent. An example of 4.01687 is discussed further. The way this data would

Table 3.3: CAN message structure 2.

Destination	Array number	Variable
String	0	Module number
	1	Voltage value
	2	Voltage value decimal portion
	3	Current value
	4	Current value decimal portion
	5	SOC value
	6	SOC value decimal portion
	7	Battery cell identifier for module
Module	0	Module number
	1	I_{all} current command
	2	I_{all} current command decimal portion
	3	V_{ref} voltage value
	4	V_{ref} voltage value decimal portion
	5	Empty
	6	Empty
	7	Start up bits command

be sent in binary can be seen in equation 3.1

$$0100.000001000101 \tag{3.1}$$

Where the first 4 bits of binary equal 4. And the 12 remaining bits equal .0168. As mentioned above, CAN messages are sent as 8-bit integers. The two bytes of data sent can be seen in equation 3.2

$$(01000000) <--> (01000101) \tag{3.2}$$

To the processor at first this looks like two integer numbers of 128 and 69 respectively, but since they are IQ, they can be unwrapped as the correct value that is sent. The code for the string controller unpacking the data can be seen in figure 3.8, as well as in the Appendix where the whole file of code is located.

```
967
968            IALL = _IQ8toF(_IQ8((MATLABmsg[1]<<8)>>8) + (_IQ8(MATLABmsg[2])>>8));
969            Vdcref = _IQ8toF(_IQ8((MATLABmsg[3]<<8)>>8) + (_IQ8(MATLABmsg[4])>>8));
970
971            startupbits = MATLABmsg[7];
972            SOC_Estimate = (startupbits>>5) & 0x0001;
973            DroopEn = (startupbits>>6) & 0x0001;
974
```

Fig. 3.8: IQ wrapping.

3.3.1 ADC implementation

An analog to digital converter (ADC) converts analog signals to a digital format. It takes an analog signal and samples the value at a certain frequency. This allows it to have one specific digital value that is updated at a fast rate. This is important because the microcontroller needs data in this format to execute its code. To set up an ADC in code there is a lot that first needs to be initialized. This initialization sets up certain parameters that the ADC follows. The most important of these parameters being the frequency at which it samples the analog signal. The ADC uses an interrupt (just like the CAN does). As the pin on the microcontroller senses a signal, it will read that data in and store it as a digital value. This digital value can be anywhere from 0 to 4096. The next step to using this digital number is to linearize it by recording the ADC value at various voltage values that are being sensed and plotting this out in MATLAB or Excel. The values for this project can be seen in table 3.4. Adding a trendline to these plots allows for an estimation to come up with an equation. The values plotted out in Excel can be seen in figure 3.9. This equation then is implemented in code to find the original voltage value that is sensed. This equation used in the code can be seen in equation 3.3.

$$Vbus = ADC_{result} - 6.8571/40.68 \tag{3.3}$$

Where ADC_{result} is the discrete value that the ADC reads in. This formula is taken from the trendline formula in figure 3.9, and then rearranged to solve for x.

3.4 Control of series connected BPMs

Since the first step of controlling multiple modules is to start them up at the same

Table 3.4: ADC Linearization.

Bus Voltage	Digital Value
0	8
5	209
10	413
15	617
20	822
25	1023
30	1227
35	1433
40	1634
45	1838
50	2040
55	2245
60	2448

time, it is necessary to have the string controller send out these commands simultaneously. This allows for each module to receive the commands at the same time and avoid issues in ramping up the duty cycle. The startup bits consist of 6 bits that are either a one or a zero. These bits and their functions are as follows.

- RSTEN. This bit controls whether the field programmable gate array (FPGA), which is located on the BPM, is in reset mode. Reset mode prevents the FPGA from executing any code or computation.

- SWEN. This bit controls the switching of the converters on the BPMs. While it is enabled, the ramping of the duty cycle is possible.

- CTLEN. This bit enables the control for closed loop which is essential for showing droop control is working.

- DIFEN. This bit enables the differential current command to account for differences among the cells in terms of charge. This allows for different charge/discharge rates.

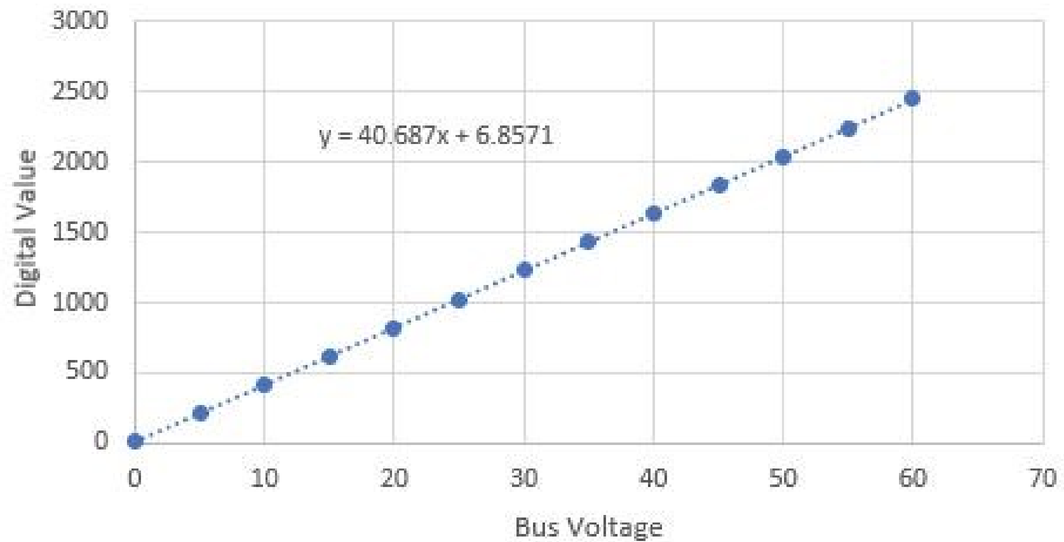


Fig. 3.9: ADC Linearization plot.

- RMPDTY. This bit enables the ramp of the duty cycle for the converters. This is essentially what enables the output voltage to be increased through the DC/DC converter.
- SOCESTEN. This bit allows for the BPM to use coulomb counting to estimate the SOC of each cell.
- DROOPEN. This bit enables the droop which allows for current regulation.

Each of these control bits are enabled one by one in a specific order to carry out the active balancing of the battery cells.

3.4.1 Droop control and delta current command

For the balancing of different cells with different SOCs, droop current needs to be implemented. See section 2.2.3. Depending on the SOC and the difference between output voltage and average voltage, the module may need more or less current than the other modules to charge or discharge. This statement can be reflected in equation 3.6.

$$I_{commonRef} = I_{All} + G_{droop} * (V_{dcref0} - V_{sec}) \tag{3.4}$$

The delta current command is a part of this equation. Delta current command is the additional current to be added or subtracted while charging or discharging. This can be seen as shown in equation 3.5 and in figure 2.5.

$$\Delta I = G_{droop} * (V_{dcref0} - V_{sec}) \tag{3.5}$$

Simplifying equation 3.6 and 3.5 for $I_{commonRef}$ gives

$$I_{commonRef} = I_{All} + \Delta I \tag{3.6}$$

An important aspect to note is that I_{All} can be positive or negative depending on the direction power is flowing. G_{droop} is different for each direction of power. Implementing this in code is done with a few conditional statements and equations. If I_{All} is positive, then G_{droop} is seen in equation 3.7.

$$G_{droop} = G_{min} \tag{3.7}$$

Where G_{min} for this topology is calculated to be .2.

If I_{All} is negative, then G_{droop} is seen in equation 3.8.

$$G_{droop} = G_{min} + (G_{min} - G_{max})/I_{allmax} * I_{all} \tag{3.8}$$

Where G_{max} must be greater than .25 due to the conditions of the system. The parameter chosen for this project for G_{max} is .5. I_{allmax} is 10 A because this is a 100 W system and there are 4 ports each at 2.5V.

$$I_{allmax} = 100/4/2.5 \tag{3.9}$$

Implementing all these concepts proves that the string controller is working to control series connected BPMs and to actively balance the battery cells. See chapter 4 for these results.

CHAPTER 4
RESULTS AND ANALYSIS

This chapter focuses on the results obtained through various tests performed and discusses how these results fulfill the previously stated objectives set in chapter 1. The chapter is broken down into 3 sections. In section 4.1 the objective of designing, developing, and building a hardware controller that can perform the functions necessary is shown and discussed. In section 4.2 the objective of developing software to realize sensing the bus voltage of the series connected battery power module (BPM)s and communicate this, along with other information through CAN is shown and discussed. In section 4.3 the objective of developing control algorithms to achieve desired SOC and cell balancing among cells, as well as droop control among the modules.

4.1 String controller

The process for designing and implementing the string controller board can be found in chapter 3. In order to make sure the board is working properly after fabrication and population; a few tests are needed. The first test can be seen in Fig. 4.1 as 12V is applied across the board's input terminals. It is essential to determine that the correct voltage is applied to the various components on the board. Whether this be 3.3 V, 5 V, etc. A couple key locations can be seen as functioning with lit light emitting diodes (LED)s, and other locations can be probed by hand using a voltage meter.

Another test that is necessary is many continuity checks. In order to verify that components are soldered correctly, and the traces are working properly, one probe is placed at the port where an input signal goes in with the other probe testing the final destination of the signal. Using a multi-meter in continuity mode, a beep should be emitted from the device if the connection is secure. An example of this can be seen in Fig. 4.2 as the input signal of an Ethernet cable is verified to reach its specific integrated circuit (IC). These tests

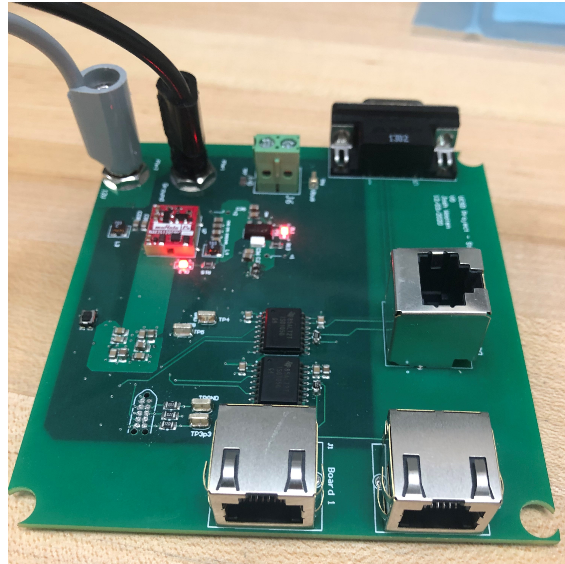


Fig. 4.1: Power test.

give confidence that the data being transmitted is received correctly by the microcontroller.

The last test for the string controller board is to make sure the ADC sensing circuit is working properly. As seen in Fig. 4.3a it is desired to read the same voltage that the power supply is set to. As seen in Fig. 4.3b it is desired to read the input voltage divided by 30, which in this example is true as the voltage meter reads .65 V. This is due to the voltage divider integrated in the circuit. The voltage divider is necessary because this voltage being sent to the microcontroller needs to be between 0 V and 3.3 V. This allows this board, theoretically, to have a 100 V input.

4.2 CAN communication and ADC sensing

The process for developing software to realize sensing the bus voltage of the series connected battery power module (BPM)s and communicate this, along with other information through CAN can be seen in chapter 3. This section focuses on the tests performed to verify the ADC sensing and CAN communication properly functions. To verify the functionality of the ADC, two different input voltages are applied across the terminal of the ADC circuit. As seen in table 4.1 a 30 V and 60 V input are applied to the terminal, and a digital value of 1229 and 2449 is read into the microcontroller. This number is then converted using equa-

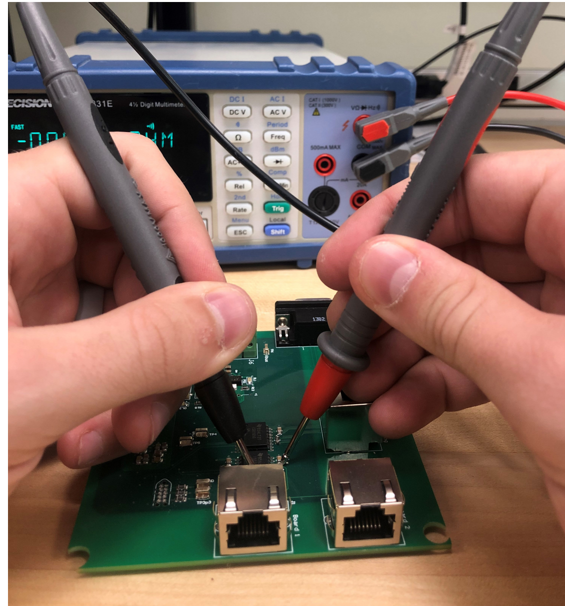


Fig. 4.2: Continuity checks.

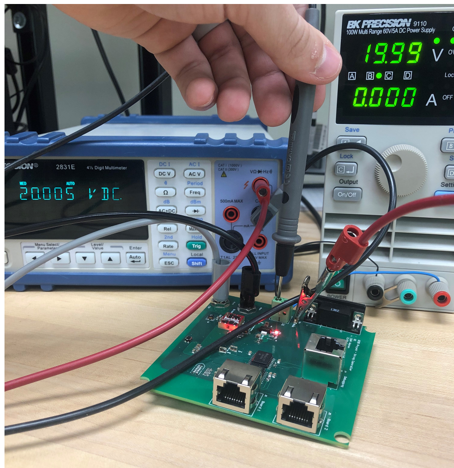
Table 4.1: ADC test.

Input voltage	Read digital result	Calculated Voltage
30	1229	30.03767
60	2449	59.9981

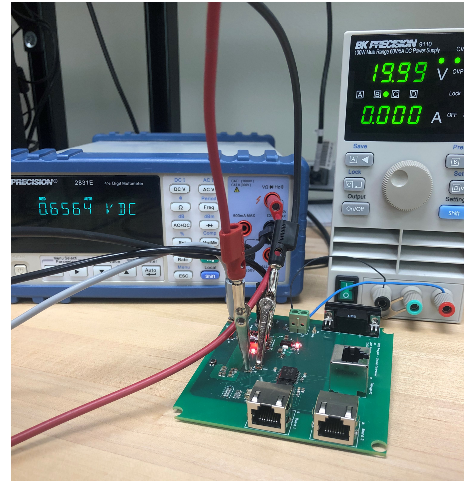
tion 3.3. This proves that the ADC is working as intended and that it is qu ite accurate. This also shows that the linearization performed on the ADC is working properly.

One challenge that came into play while implementing the ADC was the sampling frequency. With just default initialization, the sample rate was around 12kHz. Due to the fact it would send that value over CAN every time the ADC sampled in a value, it left little-to-no time for any other message to be sent/received on that CAN bus. This problem was fixed by adjusting the period of the PWM that controlled the ADC sample time. It was found that a sample rate of 1kHz works much better and allows time for other messages to get through.

To test the CAN communication, it is important to connect the string controller to both module controllers. This can be seen in fig 4.4. This allows for the string controller to send a message to one module, and a very similar but slightly different message to the



(a) ADC circuit 1.



(b) ADC circuit 2.

Fig. 4.3: ADC circuit tests

second module. Each module then receives that message through CAN, stores the data in its own respective registers, and then sends the data back. If the right data comes back from each module and is seen in the string controller's ISR, then the CAN is working great. The test results for this test can be seen in 4.5, where the data circled in orange are the two separate messages sent out, and the data circled in blue, are the messages received back. These messages reflect what is expected and prove that the CAN communication is working. It is important to note that the CAN sample rate needs to be slow enough that it does not clog the CAN network for all other messages.

4.3 System Control

4.3.1 Single module control

The first step in testing the control loop for one module is to connect the four battery cells and electronic load to the DC-DC converter. The electronic load is connected to allow for power to dissipate when the cells discharge. This converter is then connected to the string controller. The idea for this initial test it to focus on the operation for one DC-DC converter. The test set up can be seen in Fig. 4.6a and in Fig. 4.6b. The user performing the test can use MATLAB to send start up commands as mentioned in section 3.4 that

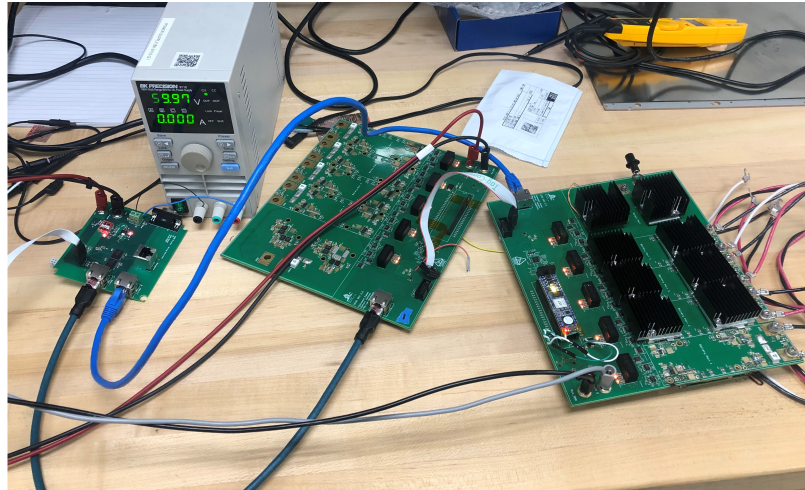


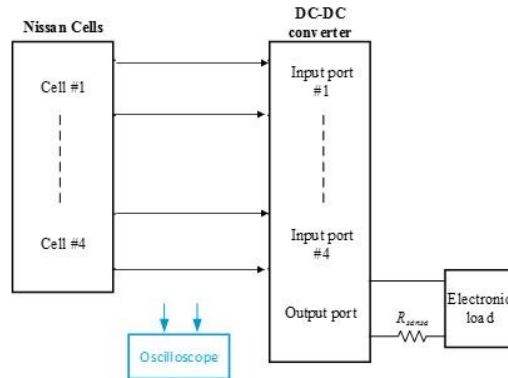
Fig. 4.4: CAN test setup.

Expression	Type	Value	Address
xCell	struct <unnamed>[1]	[{fCellVoltage=0.0,fCellCurrent=0.0,fCel...	0x0000C9AA@Data
a	unsigned int	5	0x0000CC08@Data
b	float	0.0	0x0000CC20@Data
SOCMessages.txMsgData	unsigned int[8]	[1,30,27,0,0...]	0x0000C244@Data
SOCMessages.txMsgData2	unsigned int[8]	[2,60,27,0,0...]	0x0000C24C@Data
SOCMessages.rxMsgData	int[8]	[0,0,0,0,0...]	0x0000C254@Data
AdccResultRegs.ADCRESULT0	unsigned int	2245	0x00000B40@Data
VbusCalculated	float	55.0087929	0x0000CC18@Data
c	unsigned long	122707	0x0000CC22@Data
d	unsigned int	20061	0x0000CC09@Data
canmsg.rxMsgData	unsigned int[8]	[1,60,27,0,0...]	0x0000C2D8@Data
f	int[8]	[1,30,27,0,0...]	0x0000CBB6@Data
g	int[8]	[2,60,27,0,0...]	0x0000CBEC@Data
Add new expression			

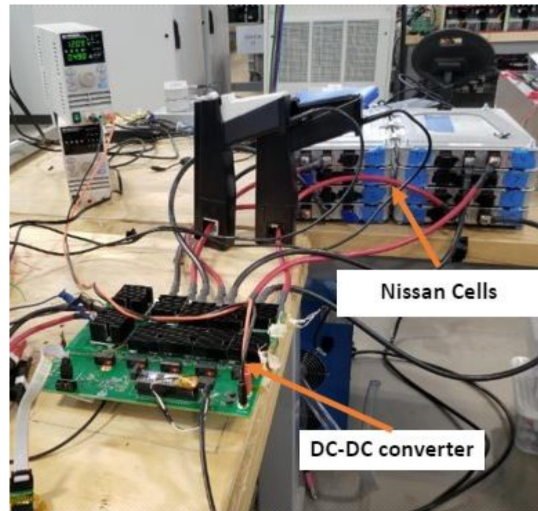
Fig. 4.5: CAN test results.

the string controller then passes on to the module controller. On the flip side, the module controller sends the cells' data to the string controller that is passed on to the user through MATLAB.

The startup commands sent from MATLAB enables switching and ramp of the duty cycle to the DC/DC converter. The results obtained can be seen in Fig. 4.7a. This is the result before the droop current loop is implemented as talked about in section 3.4. The module bus voltage is read to be 31.3 V and the current read is 6.02 A. As the droop current loop is added as seen in Fig. 4.7b the voltage is now 32.6 V and the current is 6.33A. This is exactly what is expected due to the fact that the voltage increased closer to the reference of v_{avg}, which is 33 V. See Fig. 2.5. While the modules are not connected in series, the droop will cause the bus voltage to approach closer to the reference. The increase in current



(a) One module test set up diagram.

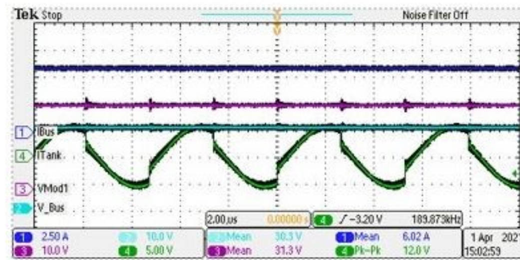


(b) One module test set up.

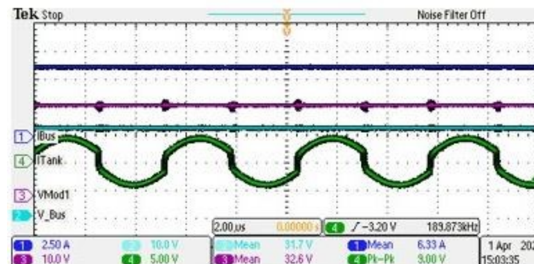
Fig. 4.6: Initial module test

that is seen is due to the difference in voltage passing through the gain G_d. This difference in voltage causes there to be a slight increase or decrease in current when droop is enabled. This addition of current can be seen in the figure as the variable idN.

To ensure that droop is fully working and validates the equations for droop presented in 3.4.1, it is necessary to perform a dc voltage reference sweep. The different voltage reference values plotted with the sensed current values shows the slope of G_{droop}. The test set up can be seen in Fig. 4.8



(a) Test without droop.



(b) Test with droop.

Fig. 4.7: Module Voltages.

G_{droop} is represented by Fig. 4.9a, 4.9b, and 4.9c, which is a plot of data shown in table 4.2. The data in the table shows droop is working as expected due to the linear relationship.

The sweep of the voltage reference shows the Gd droop to have a slope of .2 for positive and zero reference currents, and a larger slope for negative reference currents. (Around .4) The results show tests done for -10 A, 0 A, 6 A, 10 A of reference current. The data show the fluctuation of total current around these four current references depending on the voltage.

With control of a single module controller working, active cell balancing is tested. Balancing the cells consists of charging and discharging the cells at different rates based on each cells' SOC. This allows for the SOC of each cell to eventually be at the same point. This is shown in Fig. 4.10. The balancing capability results proves that the lithium-ion batteries are acceptable for second life applications.

4.3.2 Series connected module control

Once one module is controlled correctly with current control, the next step is to test

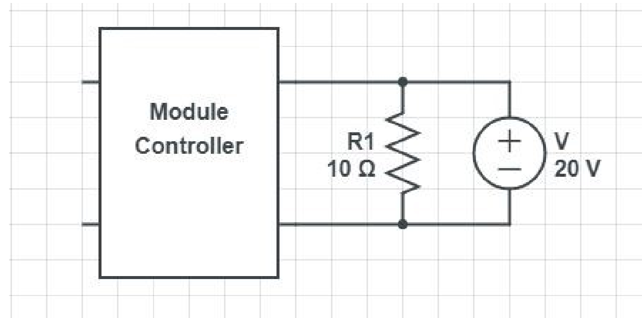
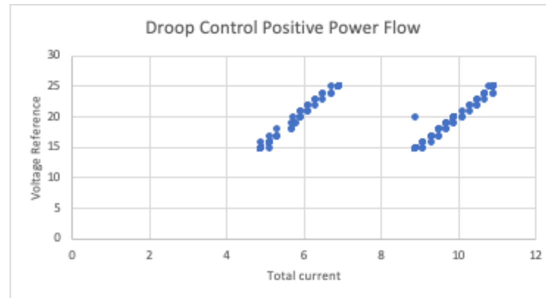


Fig. 4.8: Droop test set up.

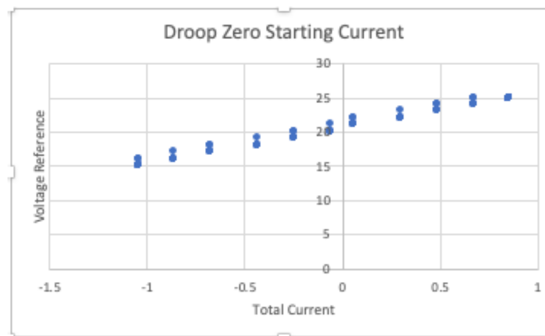
the control with two modules connected in series. The set up for this test can be seen in Fig. 4.11a and 4.11b. It is important to note that the bus voltage now consists of connecting one end to the positive output terminal of one board and the other end to the negative output terminal of the second board. The test performed here is just with two module boards, but this test works with many more boards in series.

For the series test, the battery cells lowered to 2.5 V each due to the 60 V limit on the electronic load and power supply. This allows for the output voltage to be 20 V after ramping up the duty cycle. With two modules connected in series, this leads to 40 V output instead of 64 V. The results of this connection are shown in Fig. 4.12. The three figures show tests transitioning from open loop to closed loop. 4.12a shows open loop without droop, 4.12b shows closed loop with droop enabled, and 4.12c shows closed loop without droop enabled which becomes unstable. The bus current changes as droop is enabled due to current that is dissipated by the load resistor.

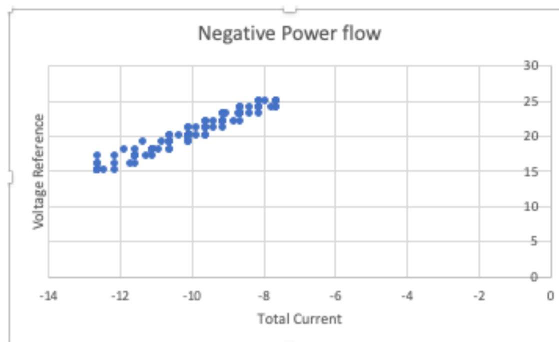
The DC voltage sweep is once again performed for a positive current reference. This can be seen in Fig. 4.13 and table 4.3. The results for series connected modules are in line with the results of a single module proving that droop still works for series connected modules. The relationship is linear, and the results show a slope of 0.2. These results once again show the capability of SOC balancing and proves the systems capability of series controlled battery powered modules. This confirms the usage of these lithium-ion batteries for second life applications.



(a) Positive reference current with droop.



(b) Zero reference current with droop.



(c) Negative reference current with droop.

Fig. 4.9: Droop results.

Table 4.2: Droop results

Reference current	Total current	Voltage reference
-10	-12.621	15
-10	-12.128	16
-10	-11.574	17
-10	-11.082	18
-10	-10.589	19
-10	-10.097	20
-10	-9.605	21
-10	-9.339	22
-10	-9.113	23
-10	-8.617	24
-10	-8.125	25
0	-1.042	15
0	-0.859	16
0	-0.675	17
0	-0.429	18
0	-0.246	19
0	-0.058	20
0	0.054	21
0	0.300	22
0	0.488	23
0	0.671	24
0	0.855	25
10	8.859	15
10	9.046	16
10	9.292	17
10	9.476	18
10	9.660	19
10	9.847	20
10	10.093	21
10	10.277	22
10	10.460	23
10	10.648	24
10	10.894	25

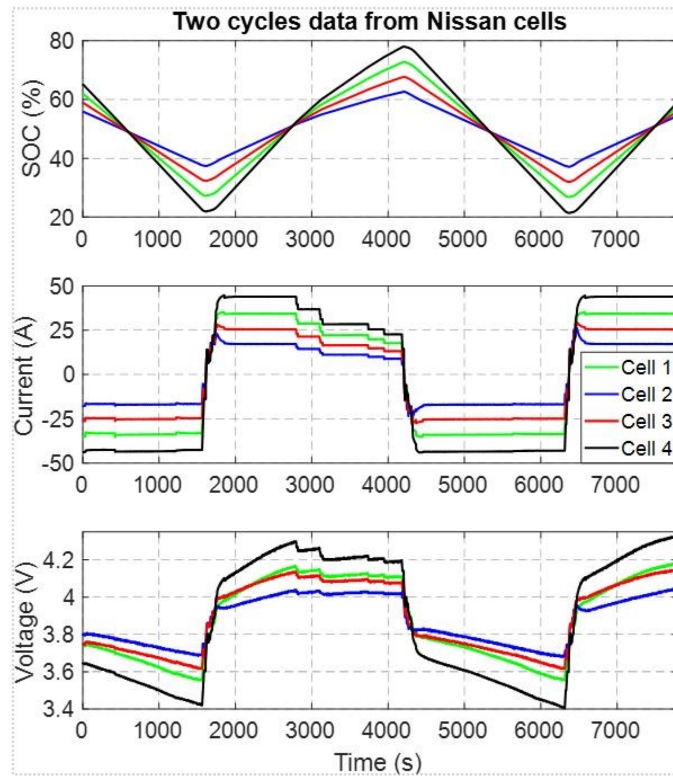
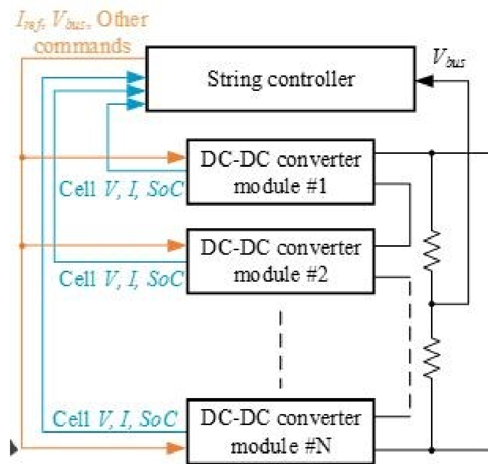


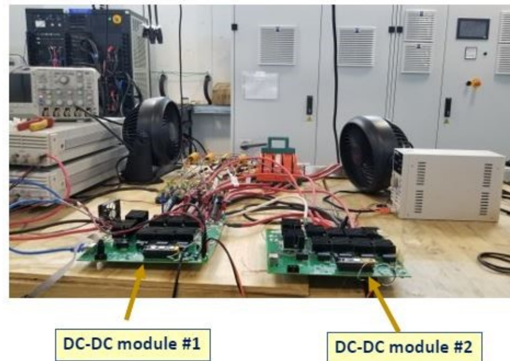
Fig. 4.10: Two cycles of cell balancing.

Table 4.3: Series control droop results

Reference current	Total current	Voltage reference
6	5.1679	15
6	5.4140	16
6	5.5351	17
6	5.5976	18
6	5.9062	19
6	6.0898	20
6	6.2734	21
6	6.4609	22
6	6.7070	23
6	7.1992	24
6	7.3203	25

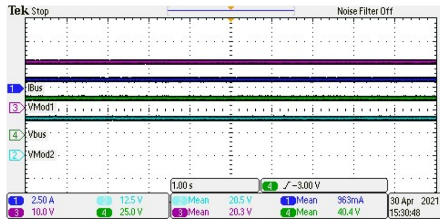


(a) Multiple module test set up diagram.

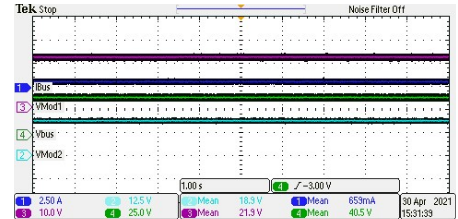


(b) Multiple module test set up.

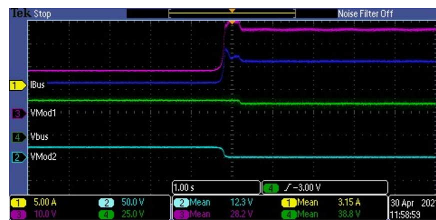
Fig. 4.11: Two module set up.



(a) Series control without droop.



(b) Series control with droop.



(c) Series control without droop with control enabled.

Fig. 4.12: Series control.

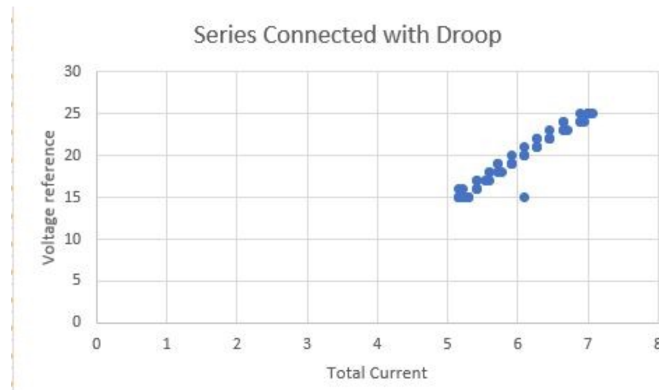


Fig. 4.13: Series control with droop sweep.

CHAPTER 5
CONCLUSION AND FUTURE WORK

As previously stated in chapter 1, the objectives for this thesis are to design, develop, and build a hardware controller that will be able to perform all necessary functions, develop software to realize sensing the bus voltage of the series connected battery power module (BPM)s and communicate this, along with other information through CAN, and to develop control algorithms to achieve desired SOC and cell balancing, as well as droop control among the modules. The results showing these completed objectives can be seen in chapter 4.

The results of the cells balancing and communicating information back and forth between the modules and string controller proves that the objectives are met. The result of active cell balancing allows for batteries to have a second life and battery packs to have a much longer lifespan. In addition, this work can be used in many other applications within the field of power electronics. Any other company that makes use of battery packs will be able to use these methods and results to use second life batteries and make the world greener.

5.1 Future work

This thesis has mostly focused on a smaller subsection of what will be the full system. The full system will be a full scale 100 kW battery pack. This thesis gives a good guide on how to continue and complete the full-scale system by using this thesis as a building block. Another thing that can be improved upon afterwards is commercializing these technologies. The world is heading toward more electric vehicles with more and more green initiative programs, and second life batteries will play an important role in this transition. Companies with similar needs will be able to use this thesis and these findings as a starting point.

One improvement that could be made to this system is adjusting and making improvements to the communication process. Even though it works fine how it is, there are always

more improvements to be found and implemented. Finding out the perfect frequency at which data should be transmitted is one example of how it could be improved. Another improvement for future work is diving into the options of using a string controller or not. Along with this, research could be done to see what more the string controller could oversee to make the system more efficient. Another improvement idea is spreading the module board into three separate boards for the three main functions of the board: power board, FPGA circuitry, and microcontroller circuitry. This design could allow for the module microcontroller to control more than four cells.

REFERENCES

[1] D. B. Yelaverthi, M. Kamel, R. Hassan, and R. Zane, "High frequency link isolated multi-port converter for active cell balancing applications," in *2019 20th Workshop on Control and Modeling for Power Electronics (COMPEL)*, 2019, pp. 1–7.

[2] M. Kamel, "Analysis and control of parallel and series connected modular battery systems," Ph.D. dissertation, Utah State University, Logan, UT, 2021.

[3] Texas Instruments. (2016) Introduction to the controller area network (can). [Online]. Available: https://www.ti.com/lit/an/sloa101b/sloa101b.pdf?ts=1618588627690&ref_url=https%253A%252F%252Fwww.google.com%252F

[4] Wikipedia. (2020) Can bus. [Online]. Available: https://en.wikipedia.org/wiki/CAN_bus

[5] M. Evzelman, M. M. Ur Rehman, K. Hathaway, R. Zane, D. Costinett, and D. Maksimovic, "Active balancing system for electric vehicles with incorporated low-voltage bus," *IEEE Transactions on Power Electronics*, vol. 31, no. 11, pp. 7887–7895, Nov 2016.

[6] Y. Li and Y. Han, "A module-integrated distributed battery energy storage and management system," *IEEE Transactions on Power Electronics*, vol. 31, no. 12, pp. 8260–8270, Dec 2016.

[7] J. A. Abu Qahouq, L. Zhang, Y. Cao, and B. Balasubramanian, "Dc-dc power converter controller for soc balancing of paralleled battery system," in *2016 IEEE Applied Power Electronics Conference and Exposition (APEC)*, March 2016, pp. 1868–1871.

[8] C. Schaef, E. Din, and J. T. Stauth, "A hybrid switched-capacitor battery management ic with embedded diagnostics for series-stacked li–ion arrays," *IEEE Journal of Solid-State Circuits*, vol. 52, no. 12, pp. 3142–3154, Dec 2017.

[9] S. Narayanaswamy, M. Kauer, S. Steinhorst, M. Lukasiewycz, and S. Chakraborty, "Modular active charge balancing for scalable battery packs," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 25, no. 3, pp. 974–987, March 2017.

[10] E. Martinez-Laserna, E. Sarasketa-Zabala, I. Villarreal Sarria, D.-I. Stroe, M. Swierczynski, A. Warnecke, J.-M. Timmermans, S. Goutam, N. Omar, and P. Rodriguez, "Technical viability of battery second life: A study from the ageing perspective," *IEEE Transactions on Industry Applications*, vol. 54, no. 3, pp. 2703–2713, 2018.

[11] D. Babu Yelaverthi, M. Muneeb Ur Rehman, and R. Zane, "Differential power processing three-port dual active bridge converter for active balancing in large battery packs," in *2018 IEEE Energy Conversion Congress and Exposition (ECCE)*, Sep. 2018, pp. 5591–5597.

[12] W. Wang and M. Preindl, "Modeling and control of a dual cell link for battery-balancing auxiliary power modules," in *2018 IEEE Applied Power Electronics Conference and Exposition (APEC)*, March 2018, pp. 3340–3345.

[13] S. Li, C. Mi, and M. Zhang, "A high efficiency low cost direct battery balancing circuit using a multi-winding transformer with reduced switch count," in *2012 Twenty-Seventh Annual IEEE Applied Power Electronics Conference and Exposition (APEC)*, Feb 2012, pp. 2128–2133.

[14] A. M. Imtiaz and F. H. Khan, ""time shared flyback converter" based regenerative cell balancing technique for series connected li-ion battery strings," *IEEE Transactions on Power Electronics*, vol. 28, no. 12, pp. 5960–5975, Dec 2013.

[15] V. Vardwaj, V. Vishakha, V. K. Jadoun, N. S. Jayalaksmi, and A. Agarwal, "Various methods used for battery balancing in electric vehicles: A comprehensive review," in *2020 International Conference on Power Electronics IoT Applications in Renewable Energy and its Control (PARC)*, 2020, pp. 208–213.

[16] Y. Yuanmao, K. W. E. Cheng, and Y. P. B. Yeung, "Zero-current switching switched-capacitor zero-voltage-gap automatic equalization system for series battery string," *IEEE Transactions on Power Electronics*, vol. 27, no. 7, pp. 3234–3242, 2012.

[17] W. Huang and J. A. Abu Qahouq, "Energy sharing control scheme for state-of-charge balancing of distributed battery energy storage system," *IEEE Transactions on Industrial Electronics*, vol. 62, no. 5, pp. 2764–2776, 2015.

[18] C. Kim, M. Kim, H. Park, and G. Moon, "A modularized two-stage charge equalizer with cell selection switches for series-connected lithium-ion battery string in an hev," *IEEE Transactions on Power Electronics*, vol. 27, no. 8, pp. 3764–3774, 2012.

[19] M. M. Ur Rehman, F. Zhang, R. Zane, and D. Maksimovic, "Control of bidirectional dc/dc converters in reconfigurable, modular battery systems," in *2017 IEEE Applied Power Electronics Conference and Exposition (APEC)*, 2017, pp. 1277–1283.

[20] R. Ayyanar, R. Giri, and N. Mohan, "Active input-voltage and load-current sharing in input-series and output-parallel connected modular dc-dc converters using dynamic input-voltage reference scheme," *IEEE Transactions on Power Electronics*, vol. 19, no. 6, pp. 1462–1473, 2004.

[21] T. Fang, X. Ruan, and C. K. Tse, "Control strategy to achieve input and output voltage sharing for input-series–output-series-connected inverter systems," *IEEE Transactions on Power Electronics*, vol. 25, no. 6, pp. 1585–1596, 2010.

[22] G. Xu, D. Sha, and X. Liao, "Decentralized inverse-droop control for input-series–output-parallel dc–dc converters," *IEEE Transactions on Power Electronics*, vol. 30, no. 9, pp. 4621–4625, 2015.

[23] Altium Designer. (2018) How to create a pcb schematic. [Online]. Available: https://resources.altium.com/p/schematic-tutorial-altium-designer-journey-thousand-pcbs

APPENDICES

APPENDIX A

String controller code

```
// ##########################################################################
// * main.c
// * Author: Josh & Mohamed
// * Created on: May, 2019
// * Modified by Josh W 2020
// ##########################################################################
#include "global.h"
#include "CONV_ControlLaw_shared_data.h"
#include "F28x_Project.h"
#pragma CODE_SECTION(CONV_timeline_isr, ".TI.ramfunc");
extern uint16_t RamfuncsLoadStart, RamfuncsLoadSize, RamfuncsRunStart;


// Define some local functions

interrupt void CONV_timeline_isr(void);
interrupt void cpuTimer1ISR(void);
interrupt void cpuTimer2ISR(void);
interrupt void canaISR(void);
interrupt void canbISR(void);
interrupt void CONV_adc_isr(void);
void ATREX_CAN_Communication(void);
void OneSecondHandle(void);
void ONeSecondSOCEstimator(void);
void SOCInitialization(void);
void initADC();
```

```
void initADCSOC();
void initEPWM();

uint16_t EnableSense = 0;
int ProtectionCurrent = 0;
float CurrentInputInFloat = 0;

_iq  VuMinusVariable = 0;
_iq  VLVariable = 0;
_iq  VbusVariable = 0;

uint16_t BadCommunicationCounter = 0;
uint16_t SOC_Initialize = 0;
uint16_t Timer1Flag = 0;
uint16_t SOC_Estimate = 0;
uint16_t count = 0;
uint16_t DebugCount = 0;

uint16_t counter0 = 0;
uint16_t counter1 = 0;
uint16_t counter2 = 0;
uint16_t counter3 = 0;

float VbusCalculated = 0;
float Vmodule = 0;
float Iref = 30.0f;

uint16_t a = 5;
float b = 0;
uint32_t c = 0;
uint16_t d = 0;
float module1Cell1[8];
```

```
float module1Cell2[8];
float module1Cell3[8];
float module1Cell4[8];
float module2Cell1[8];
float module2Cell2[8];
float module2Cell3[8];
float module2Cell4[8];
int startupbits;

// startup commands
//int16_t DiffEn = 0;
//int16_t RST = 1;
//int16_t CTLEn = 0;
//int16_t SWEn = 0;
//int16_t rampDuty = 0;
int16_t SOC_Est_En = 0;
int16_t DroopEn = 0;

//Josh's variables for CAN
int16_t MATLABmsg[8];
//uint32_t c = 0;
float IALL = 0.0f;
float Vdcref = 0.0f;



uint16_t myResults[100];
uint16_t index2 = 0;
uint16_t testResult = 0;
uint16_t testResultMean = 0;

// Added in for ADC ISR
#define RESULTS_BUFFER_SIZE  256
```

```
uint16_t adcCResults[RESULTS_BUFFER_SIZE]; // Buffer for results
uint16_t index;                            // Index into result buffer
volatile uint16_t bufferFull;              // Flag to indicate buffer is full


//###############################################################################
// ------ main function ---------------------------------------------------------
// * system initialization
// -----------------------------------------------------------------------------
void main(void)
{
    //==========================================================================
    //---------- Step 1. Initialize System Control -----------------------------
    memcpy((uint16_t *)&RamfuncsRunStart,(uint16_t *)&RamfuncsLoadStart, (unsigned
         long)&RamfuncsLoadSize);
    InitSysCtrl();    // PLL, Flash, WatchDog, enable Peripheral Clocks- External
        oscillator, 100MHz system clock
    // Initialize GPIO
    InitGpio();
    //==========================================================================
    //---------- Step 2. Initialize GPIO ---------------------------------------
    OpenLoopGPIOsNewPack();
    DELAY_US(500000); // This Delay is critical with the LTC2955 Pushbutton
        Control. The Blanking time ignores and seems to oppose/invert ;any changes
        to PB or Kill during that half second.
    //==========================================================================
    //---------- Step 3. Initialize PIE vector table ---------------------------
    // Disable CPU interrupts and clear all CPU interrupt flags:
    DINT;
    IER = 0x0000;
    IFR = 0x0000;
```

```
                  // IER and IFR are special registers and are defined by "
                      cregister".
                  // The compiler handles cregisters differently because they
                      require special assembly output.
                  // They are not defined similarly as other registers in the
                      header files.

InitPieCtrl();    // Initialize PIE control registers to their default state
InitPieVectTable(); // Initialize the PIE vector table with pointers to the
    shell Interrupt
// * Re-mapped interrupts to ISR functions found within this file:
EALLOW;
PieVectTable.TIMER0_INT = &CONV_timeline_isr;
//PieVectTable.TIMER1_INT = &cpuTimer1ISR;
//PieVectTable.TIMER2_INT = &cpuTimer2ISR;
PieVectTable.CANA0_INT = &canaISR;
PieVectTable.CANB0_INT = &canbISR;
PieVectTable.ADCC1_INT = &CONV_adc_isr; // Set this to C1 ADC which is pin 18
    for our microcontroller
EDIS;
//===========================================================================
//---------- Step 4.0. Initialize CLA --------------------------------------
//InitializeCLA();
//===========================================================================
//---------- Step 5. Initialize all the Device Peripherals ------------------
// * Configure CPU-Timer 0 to interrupt at 10kHz (with 100Mhz CPU):
InitializeTimers();
// * Initialize EPWM module:
//CONV_InitEPwm();
initEPWM(); // Start up example code implemented by Josh
// * Initialize ADC module:
CONV_InitAdc();
```

```
//initADC(); // Start up example code implemented by Josh
// * Initialize ADC SOC:
initADCSOC();
// Initialize the state structure SOC ESTIMATION
vBmsInit( xCell );
// * Initialize CAN bus:
CONV_InitECana();
CONV_InitECanb();
// * Initialize global variables:
InitGlobalVariable();
//============================================================================
//---------- Step 6. Enable interrupts --------------------------------------
// * Enable specific interrupts in the PIE table:
PieCtrlRegs.PIECTRL.bit.ENPIE = 1; //*0. Enable the PIE block
PieCtrlRegs.PIEIER1.bit.INTx3 = 1; //*1. Enable ADCC1 in the PIE: Group 1
    interrupt 3
PieCtrlRegs.PIEIER1.bit.INTx7 = 1; //*1. Enable TINT0 in the PIE: Group 1
    interrupt 7
PieCtrlRegs.PIEIER9.bit.INTx5 = 1; //*1. Enable CANA0 in the PIE: Group 9
    interrupt 5
PieCtrlRegs.PIEIER9.bit.INTx7 = 1; //*1. Enable CANB0 in the PIE: Group 9
    interrupt 7

// * Enable interrupt:
IER |= M_INT1; // Enable group 1 interrupts - ADC interrupts - External
    interrupt 1 and 2 - and Timer 0
//IER |= M_INT13; // Enable group 14 interrupts - Timer1
//IER |= M_INT14; // Enable group 14 interrupts - Timer2
IER |= M_INT9; // Enable group 9 interrupts - CANA0 and CANB0 interrupt
// * Enable global Interrupts and higher priority real-time debug events:
EINT;  // Enable Global interrupt INTM
ERTM;  // Enable Global realtime interrupt DBGM
```

```
    //===========================================================================
    //---------- Step 7. User specified code ----------------------------------
    //StartTimer(0);
    SOC_Estimate = 0;
    DebugCount = 0;
    EALLOW;
    // Sync ePWM
    CpuSysRegs.PCLKCR0.bit.TBCLKSYNC = 1;

    while(1)
    {
//          // SOC Functions/1 Second interrupt Enabled
//              if (Timer1Flag == 1)
//              {
//                  OneSecondHandle();
//                  // Clear the timer interrupt here!
//                  Timer1Flag = 0;
////                  if (count > 1){count = 2;SOC_Estimate = 1;DebugCount++;}
////                  PieCtrlRegs.PIEACK.all = PIEACK_GROUP1;
//              }

              //
                    // Start ePWM
                    //
                    EPwm1Regs.ETSEL.bit.SOCAEN = 1; // Enable SOCA
                    EPwm1Regs.TBCTL.bit.CTRMODE = 0; // Unfreeze, and enter up
                        count mode

                    //
                    // Wait while ePWM causes ADC conversions, which then cause
                        interrupts,
```

```
                    // which fill the results buffer, eventually setting the
                        bufferFull
                    // flag
                    //
                    while(!bufferFull)
                    {
                    }
                    bufferFull = 0; //clear the buffer full flag

                    //
                    // Stop ePWM
                    //
//                      EPwm1Regs.ETSEL.bit.SOCAEN = 0; // Disable SOCA
//                      EPwm1Regs.TBCTL.bit.CTRMODE = 3; // Freeze counter
//
//                      //
//                      // Software breakpoint. At this point, conversion results
    are stored in
//                      // adcAResults.
//                      //
//                      // Hit run again to get updated conversions.
//                      //
//                       ESTOP0;

    }
}
//###############################################################################
interrupt void CONV_adc_isr(void)
{
    _iq8 IQ_Iref, IQ_Vref;
    c++;
```

```
// This is from example code. Basic ISR functionality



// Add the latest result to the buffer
// ADCRESULT0 is the result register of SOC0
adcCResults[index++] = AdccResultRegs.ADCRESULT0;
VbusCalculated = ((AdccResultRegs.ADCRESULT0-6.8571)/40.687); //equation
    calculated from linearizing ADC results and plotting them to find trendline
     formula

//
// Set the bufferFull flag if the buffer is full
//
if(RESULTS_BUFFER_SIZE <= index)
{
    index = 0;
    bufferFull = 1;
}

//
// Clear the interrupt flag
//
AdccRegs.ADCINTFLGCLR.bit.ADCINT1 = 1;

//
// Check if overflow has occurred
//
if(1 == AdccRegs.ADCINTOVF.bit.ADCINT1)
{
    AdccRegs.ADCINTOVFCLR.bit.ADCINT1 = 1; //clear INT1 overflow flag
    AdccRegs.ADCINTFLGCLR.bit.ADCINT1 = 1; //clear INT1 flag
}
```

```
    //
    // Acknowledge the interrupt
    //
    PieCtrlRegs.PIEACK.all = PIEACK_GROUP1;

    Vmodule = VbusCalculated / 2; // Divide by the number of modules connected in
        series
    Vmodule = Vdcref;

//    startupbits = (DroopEn<<6) + (SOC_Est_En<<5) + (rampDuty<<4) + (DiffEn<<3) +
    (RST<<2) + (CTLEn<<1) + SWEn;
   // Test sending data through CAN


    IQ_Iref = _IQ8(IALL); //Brooks changed this from Iref to IALL due to MATLAB
        sending the command
    IQ_Vref = _IQ8(Vmodule);

    // Brooks changed all of these to canmsg_string (declarations in global files)
    canmsg_string.txMsgData[0] = 1; // Message ID Send to Module 1
    canmsg_string.txMsgData[1] = (uint16_t)IQ_Iref>>8; //Iall current that is
        currently changed by the user
    canmsg_string.txMsgData[2] = (uint16_t)IQ_Iref; //Voltage of one cell
    canmsg_string.txMsgData[3] = (uint16_t)IQ_Vref>>8; //this is start up command
    canmsg_string.txMsgData[4] = (uint16_t)IQ_Vref;
    canmsg_string.txMsgData[5] = 0;
    canmsg_string.txMsgData[6] = 0;
    canmsg_string.txMsgData[7] = startupbits;
    CAN_sendMessage(CANB_BASE, 1, MSG_DATA_LENGTH, canmsg_string.txMsgData);

    canmsg_string.txMsgData2[0] = 2; // This is a test varriable
```

```
    canmsg_string.txMsgData2[1] = (uint16_t)IQ_Iref>>8; //Iall current that is
        currently changed by the user
    canmsg_string.txMsgData2[2] = (uint16_t)IQ_Iref; //Voltage of one cell if they
         are equal
    canmsg_string.txMsgData2[3] = (uint16_t)IQ_Vref>>8; //The rest are start up
        commands
    canmsg_string.txMsgData2[4] = (uint16_t)IQ_Vref;
    canmsg_string.txMsgData2[5] = 0;
    canmsg_string.txMsgData2[6] = 0;
    canmsg_string.txMsgData2[7] = startupbits;
    CAN_sendMessage(CANB_BASE, 1, MSG_DATA_LENGTH, canmsg_string.txMsgData2);
}

//################################################################################
// ------ timeline interrupt function ------------------------------------------
// * interrupt generated every 100us
// ----------------------------------------------------------------------------
interrupt void CONV_timeline_isr(void)
{
    // The flag is cleared in OneSecondHandle function
    Timer1Flag = 1;
    count++;
    PieCtrlRegs.PIEACK.all = PIEACK_GROUP1;
//     ////debugx.timer0[1] = 5999 - CpuTimer0.RegsAddr->TIM.all;
//     //========================================================================
//     // Acknowledge this interrupt to receive more interrupts from group 1
//     PieCtrlRegs.PIEACK.all = PIEACK_GROUP1;
//     return;
//
}

//################################################################################
```

```
// ------ cpuTimer1ISR interrupt function
    ------------------------------------------
// * interrupt generated every 1 s
// -----------------------------------------------------------------------------
interrupt void cpuTimer1ISR(void)
{
// // Whatever!
//   Atrexmsg.txMsgData[0]++;
//   Atrexmsg.txMsgData[3]++;
//   EnableSense++;
//     CAN_sendMessage(CANA_BASE, Atrex_Tx_SOC_SOH, MSG_DATA_LENGTH, Atrexmsg.
    txMsgData);// Transmit arbitrary response debugging
//
//     CAN_sendMessage(CANA_BASE, Atrex_Tx_Vout, MSG_DATA_LENGTH, Atrexmsg.
    txMsgData);// Transmit arbitrary response debugging
//
//
//   CAN_sendMessage(CANA_BASE, Atrex_Tx_Capacity, MSG_DATA_LENGTH, Atrexmsg.
    txMsgData);// Transmit arbitrary response debugging
//
//     CAN_sendMessage(CANA_BASE, Atrex_Tx_Currents, MSG_DATA_LENGTH, Atrexmsg.
    txMsgData);// Transmit arbitrary response debugging
//
//
//     CAN_sendMessage(CANA_BASE, Atrex_Tx_Temperature, MSG_DATA_LENGTH, Atrexmsg.
    txMsgData);// Transmit arbitrary response debugging




//   uint16_t MOduleIndex = 0;
//   for(MOduleIndex=0; MOduleIndex<10; MOduleIndex++)
```

```
//    {
//        SOCMessages.txMsgData[0] = 1; // This is SOC information
//        SOCMessages.txMsgData[1] = 0;// This is the module average
//        SOCMessages.txMsgData[2] = (uint16_t)Pack_IQ.CellSOC_CAN[3*MOduleIndex
    ]>>8; // This is the first cell in this Module
//        SOCMessages.txMsgData[3] = (uint16_t)Pack_IQ.CellSOC_CAN[3*MOduleIndex
    ];// This is the first cell in this Module
//        SOCMessages.txMsgData[4] = (uint16_t)Pack_IQ.CellSOC_CAN[3*MOduleIndex +
    1]>>8; // This is the second cell in this Module
//        SOCMessages.txMsgData[5] = (uint16_t)Pack_IQ.CellSOC_CAN[3*MOduleIndex +
    1]; // This is the second cell in this Module
//        SOCMessages.txMsgData[6] = (uint16_t)Pack_IQ.CellSOC_CAN[3*MOduleIndex +
    2]>>8; // This is the third cell in this Module
//        SOCMessages.txMsgData[7] = (uint16_t)Pack_IQ.CellSOC_CAN[3*MOduleIndex +
    2]; // This is the third cell in this Module
//        CAN_sendMessage(CANA_BASE, MOduleIndex+14, MSG_DATA_LENGTH, SOCMessages.
    txMsgData);
//    }

}

//###############################################################################
// ------ cpuTimer2ISR interrupt function
    ------------------------------------------
// * interrupt generated every 1 s
// -----------------------------------------------------------------------------
interrupt void cpuTimer2ISR(void)
{

}

interrupt void canbISR(void)
```

```
{
    d++;
    canbmsg.status = CAN_getInterruptCause(CANB_BASE); // Read the CAN-B interrupt
         status to find the cause of the interrupt
    CAN_readMessage(CANB_BASE, canbmsg.status, canbmsg.rxMsgData);

    int cellIndex = (canbmsg.rxMsgData[0]<<2) + canbmsg.rxMsgData[7];

//    xCell[cellIndex].fCellVoltage = _IQ12toF(_IQ12((canbmsg.rxMsgData[1]<<8)>>8)
    + (_IQ12(canbmsg.rxMsgData[2])>>12));
    xCell[cellIndex].fCellVoltage = _IQ12toF(_IQ12((((int16_t)canbmsg.rxMsgData
        [1])<<8) + (int16_t)canbmsg.rxMsgData[2])>>12);
    xCell[cellIndex].fCellCurrent = _IQ8toF (_IQ8 ((((int16_t)canbmsg.rxMsgData
        [3])<<8) + (int16_t)canbmsg.rxMsgData[4])>>8);
    xCell[cellIndex].fCellSoc    = _IQ15toF(_IQ15((((int16_t)canbmsg.rxMsgData[5])
        <<8) + (int16_t)canbmsg.rxMsgData[6])>>15);

    if(canbmsg.rxMsgData[7]==0)
        {
            counter0++;
        }
    if(canbmsg.rxMsgData[7]==1)
        {
            counter1++;
        }
    if(canbmsg.rxMsgData[7]==2)
        {
            counter2++;
        }
    if(canbmsg.rxMsgData[7]==3)
    {
        counter3++;
```

```
    }
//    if(canmsg_string.rxMsgData[0]==1) //if the identifier is 1, the data is from
    module 1
//    {
//        if(canmsg_string.rxMsgData[7]==1) //if identifier in 8th spot is 1, it is
     from cell 1
//        {
//            module1Cell1[0] = canmsg_string.rxMsgData[0];
//            module1Cell1[1] = canmsg_string.rxMsgData[1];
//            module1Cell1[2] = canmsg_string.rxMsgData[2];
//            module1Cell1[3] = canmsg_string.rxMsgData[3];
//            module1Cell1[4] = canmsg_string.rxMsgData[4];
//            module1Cell1[5] = canmsg_string.rxMsgData[5];
//            module1Cell1[6] = canmsg_string.rxMsgData[6];
//            module1Cell1[7] = canmsg_string.rxMsgData[7];
//        }
//        if(canmsg_string.rxMsgData[7]==2) //if identifier in 8th spot is 1, it is
     from cell 2
//        {
//            module1Cell2[0] = canmsg_string.rxMsgData[0];
//            module1Cell2[1] = canmsg_string.rxMsgData[1];
//            module1Cell2[2] = canmsg_string.rxMsgData[2];
//            module1Cell2[3] = canmsg_string.rxMsgData[3];
//            module1Cell2[4] = canmsg_string.rxMsgData[4];
//            module1Cell2[5] = canmsg_string.rxMsgData[5];
//            module1Cell2[6] = canmsg_string.rxMsgData[6];
//            module1Cell2[7] = canmsg_string.rxMsgData[7];
//        }
//        if(canmsg_string.rxMsgData[7]==3) //if identifier in 8th spot is 1, it is
     from cell 3
//        {
//            module1Cell3[0] = canmsg_string.rxMsgData[0];
```

```
//          module1Cell3[1] = canmsg_string.rxMsgData[1];
//          module1Cell3[2] = canmsg_string.rxMsgData[2];
//          module1Cell3[3] = canmsg_string.rxMsgData[3];
//          module1Cell3[4] = canmsg_string.rxMsgData[4];
//          module1Cell3[5] = canmsg_string.rxMsgData[5];
//          module1Cell3[6] = canmsg_string.rxMsgData[6];
//          module1Cell3[7] = canmsg_string.rxMsgData[7];
//      }
//      if(canmsg_string.rxMsgData[7]==4) //if identifier in 8th spot is 1, it is
     from cell 4
//      {
//          module1Cell4[0] = canmsg_string.rxMsgData[0];
//          module1Cell4[1] = canmsg_string.rxMsgData[1];
//          module1Cell4[2] = canmsg_string.rxMsgData[2];
//          module1Cell4[3] = canmsg_string.rxMsgData[3];
//          module1Cell4[4] = canmsg_string.rxMsgData[4];
//          module1Cell4[5] = canmsg_string.rxMsgData[5];
//          module1Cell4[6] = canmsg_string.rxMsgData[6];
//          module1Cell4[7] = canmsg_string.rxMsgData[7];
//      }
//   }
//   else if(canmsg_string.rxMsgData[0]==2) //if the identifier is 2, the data is
    from module 2
//   {
//      if(canmsg_string.rxMsgData[7]==1) //if identifier in 8th spot is 1, it is
     from cell 1
//      {
//          module2Cell1[0] = canmsg_string.rxMsgData[0];
//          module2Cell1[1] = canmsg_string.rxMsgData[1];
//          module2Cell1[2] = canmsg_string.rxMsgData[2];
//          module2Cell1[3] = canmsg_string.rxMsgData[3];
//          module2Cell1[4] = canmsg_string.rxMsgData[4];
```

```
//          module2Cell1[5] = canmsg_string.rxMsgData[5];
//          module2Cell1[6] = canmsg_string.rxMsgData[6];
//          module2Cell1[7] = canmsg_string.rxMsgData[7];
//      }
//      if(canmsg_string.rxMsgData[7]==2) //if identifier in 8th spot is 1, it is
     from cell 2
//      {
//          module2Cell2[0] = canmsg_string.rxMsgData[0];
//          module2Cell2[1] = canmsg_string.rxMsgData[1];
//          module2Cell2[2] = canmsg_string.rxMsgData[2];
//          module2Cell2[3] = canmsg_string.rxMsgData[3];
//          module2Cell2[4] = canmsg_string.rxMsgData[4];
//          module2Cell2[5] = canmsg_string.rxMsgData[5];
//          module2Cell2[6] = canmsg_string.rxMsgData[6];
//          module2Cell2[7] = canmsg_string.rxMsgData[7];
//      }
//      if(canmsg_string.rxMsgData[7]==3) //if identifier in 8th spot is 1, it is
     from cell 3
//      {
//          module2Cell3[0] = canmsg_string.rxMsgData[0];
//          module2Cell3[1] = canmsg_string.rxMsgData[1];
//          module2Cell3[2] = canmsg_string.rxMsgData[2];
//          module2Cell3[3] = canmsg_string.rxMsgData[3];
//          module2Cell3[4] = canmsg_string.rxMsgData[4];
//          module2Cell3[5] = canmsg_string.rxMsgData[5];
//          module2Cell3[6] = canmsg_string.rxMsgData[6];
//          module2Cell3[7] = canmsg_string.rxMsgData[7];
//      }
//      if(canmsg_string.rxMsgData[7]==4) //if identifier in 8th spot is 1, it is
     from cell 4
//      {
//          module2Cell4[0] = canmsg_string.rxMsgData[0];
```

```
//          module2Cell4[1] = canmsg_string.rxMsgData[1];
//          module2Cell4[2] = canmsg_string.rxMsgData[2];
//          module2Cell4[3] = canmsg_string.rxMsgData[3];
//          module2Cell4[4] = canmsg_string.rxMsgData[4];
//          module2Cell4[5] = canmsg_string.rxMsgData[5];
//          module2Cell4[6] = canmsg_string.rxMsgData[6];
//          module2Cell4[7] = canmsg_string.rxMsgData[7];
//      }
//   }

//
//
//   //   Atrexmsg.status = CAN_getInterruptCause(CANA_BASE); // Read the CAN-A
    interrupt status to find the cause of the interrupt
//   //
//   //  CAN_clearInterruptStatus(CANA_BASE, Atrexmsg.status);
//   //
//   //  CAN_clearGlobalInterruptStatus(CANA_BASE, CAN_GLOBAL_INT_CANINT0); //
    Clear the global interrupt flag for the CAN interrupt line
//   //
//   //  Interrupt_clearACKGroup(INTERRUPT_ACK_GROUP9);  // Acknowledge this
    interrupt located in group 9
//
//      //BMSMEM_handle u = &mem;
//       canmsg_string.status = CAN_getInterruptCause(CANB_BASE); // Read the CAN
    -B interrupt status to find the cause of the interrupt
//       if(canmsg_string.status >= Module_Mailbox_1 && canmsg_string.status <=
    Module_Mailbox_10) // This is a message from the module we are looking for
//       {
//              CAN_readMessage(CANB_BASE, canmsg_string.status, canmsg_string.
    rxMsgData);
//              switch(canmsg_string.rxMsgData[0])
```

```
//              {
//                  case CAN_Vcell: // Store Vcell into corresponding Module //
    IF USING THIS CODE, MUST CHECK if canmsg or canmsg_string
//  //                  xCell[canmsg.status*3-3].fCellVoltage = ADC_K_Vi*(((
    canmsg.rxMsgData[2] <<8) + canmsg.rxMsgData[3])*1.0f);
//  //                  xCell[canmsg.status*3-2].fCellVoltage = ADC_K_Vi*(((
    canmsg.rxMsgData[4] <<8) + canmsg.rxMsgData[5])*1.0f);
//  //                  xCell[canmsg.status*3-1].fCellVoltage = ADC_K_Vi*(((
    canmsg.rxMsgData[6] <<8) + canmsg.rxMsgData[7])*1.0f);
//                      xCell[0].fCellVoltage = ADC_K_Vi*(((canmsg.rxMsgData[2]
    <<8) + canmsg.rxMsgData[3])*2.0f);
//                      canmsg.Icell[0] = (((canmsg.rxMsgData[4] <<8) + canmsg.
    rxMsgData[5]));
//                      CurrentInputInFloat = ADC_K_Ix*canmsg.Icell[0] ;
//                      xCell[0].fCellCurrent = (-ADC_K_Ix*(canmsg.Icell[0]) +
    canmsg.FloatOffset);
//                  break;
//              }
//             // Now let's make sure we estimate the SOC
//             if (SOC_Estimate == 0)
//             {
//                 vBmsEstInit( xCell, 0 );
//                 SOC_Estimate++;
//  //              Pack_IQ.CellSOC_CAN[0] = _IQ15(xCell[ 0 ].fCellSoc);
//  //              Pack_IQ.CellSOC_CAN_V2[0] = _IQ14(xCell[ 0 ].fCellSoc);
//  //              SOCMessages.txMsgData[0] = 1; // This is SOC information
//  //              SOCMessages.txMsgData[1] = 0;// This is the module average
//  //              SOCMessages.txMsgData[2] = (uint16_t)Pack_IQ.CellSOC_CAN
    [0]>>8; // This is the first cell in this Module
//  //              SOCMessages.txMsgData[3] = (uint16_t)Pack_IQ.CellSOC_CAN
    [0];// This is the first cell in this Module
```

```
//   //               SOCMessages.txMsgData[4] = 0; // This is the second cell in
    this Module
//   //               SOCMessages.txMsgData[5] = 0; // This is the second cell in
    this Module
//   //               SOCMessages.txMsgData[6] = 0; // This is the third cell in
    this Module
//   //               SOCMessages.txMsgData[7] = 0; // This is the third cell in
    this Module
//   //               CAN_sendMessage(CANA_BASE, 14, MSG_DATA_LENGTH, SOCMessages.
    txMsgData); // We are sending from mailbox #14
//              }
//              else
//              {
//                  // Step in time
//                  vBmsEstStep( xCell, 0);
//                  // Transmit the estimated SOC
//              }
//          }
//   //     else if(canmsg.status == ComputerDebug_Mailbox_13) // Computer Debug
//   //     {
//   //         CAN_readMessage(CANA_BASE, ComputerDebug_Mailbox_13, canmsg.
    rxMsgData);
//   //     }
//          else   // Unexpected message received
//          {
//          }
//          Pack_IQ.CellSOC_CAN[0] = _IQ15(xCell[ 0 ].fCellSoc);
//          Pack_IQ.CellSOC_CAN_V2[0] = _IQ14(xCell[ 0 ].fCellSoc);
//          SOCMessages.txMsgData[0] = 1; // This is SOC information
//          SOCMessages.txMsgData[1] = 0;// This is the module average
//          SOCMessages.txMsgData[2] = (uint16_t)Pack_IQ.CellSOC_CAN[0]>>8; // This
    is the first cell in this Module
```

```
//        SOCMessages.txMsgData[3] = (uint16_t)Pack_IQ.CellSOC_CAN[0];// This is
    the first cell in this Module
//        SOCMessages.txMsgData[4] = 0; // This is the second cell in this Module
//        SOCMessages.txMsgData[5] = 0; // This is the second cell in this Module
//        SOCMessages.txMsgData[6] = 0; // This is the third cell in this Module
//        SOCMessages.txMsgData[7] = 0; // This is the third cell in this Module
//        CAN_sendMessage(CANB_BASE, 14, MSG_DATA_LENGTH, SOCMessages.txMsgData);
    // We are sending from mailbox #14
        DebugCount++;
           CAN_clearInterruptStatus(CANB_BASE, canbmsg.status); // Clear the
               interrupt from message object

           CAN_clearGlobalInterruptStatus(CANB_BASE, CAN_GLOBAL_INT_CANINT0); //
               Clear the global interrupt flag for the CAN interrupt line

           Interrupt_clearACKGroup(INTERRUPT_ACK_GROUP9);  // Acknowledge this
               interrupt located in group 9
}

/*******************************************************************************
 *  FUNCTION      : __interrupt void canbISR(void)
 *     return     : void
 *     arg        : void
 *  Created by    : Mohamed Kamel
 *  Date created  : 05/28/2018
 *  Description   : This ISR is intended for responding to ATREX, it currently
     only sends an arbitrary response for testing purposes
 *                :
 *                :
 *  Notes         :
 ******************************************************************************/
interrupt void canaISR(void)
```

```
{
//   c++;
//   Atrexmsg.status = CAN_getInterruptCause(CANA_BASE); // Read the CAN-A
    interrupt status to find the cause of the interrupt
//
//  CAN_clearInterruptStatus(CANA_BASE, Atrexmsg.status);
//
//  CAN_clearGlobalInterruptStatus(CANA_BASE, CAN_GLOBAL_INT_CANINT0); // Clear
    the global interrupt flag for the CAN interrupt line
//
//  Interrupt_clearACKGroup(INTERRUPT_ACK_GROUP9);  // Acknowledge this interrupt
    located in group 9
//
//   //BMSMEM_handle u = &mem;
//    canmsg.status = CAN_getInterruptCause(CANA_BASE); // Read the CAN-A
    interrupt status to find the cause of the interrupt
//    if(canmsg.status >= Module_Mailbox_1 && canmsg.status <= Module_Mailbox_10)
    // This is a message from the module we are looking for
//    {
//           CAN_readMessage(CANA_BASE, canmsg.status, canmsg.rxMsgData);
//           switch(canmsg.rxMsgData[0])
//            {
//                 case CAN_Vcell: // Store Vcell into corresponding Module
////                     xCell[canmsg.status*3-3].fCellVoltage = ADC_K_Vi*(((canmsg
    .rxMsgData[2] <<8) + canmsg.rxMsgData[3])*1.0f);
////                     xCell[canmsg.status*3-2].fCellVoltage = ADC_K_Vi*(((canmsg
    .rxMsgData[4] <<8) + canmsg.rxMsgData[5])*1.0f);
////                     xCell[canmsg.status*3-1].fCellVoltage = ADC_K_Vi*(((canmsg
    .rxMsgData[6] <<8) + canmsg.rxMsgData[7])*1.0f);
//                     xCell[0].fCellVoltage = ADC_K_Vi*(((canmsg.rxMsgData[2] <<8)
     + canmsg.rxMsgData[3])*2.0f);
```

```
//                      canmsg.Icell[0] = (((canmsg.rxMsgData[4] <<8) + canmsg.
    rxMsgData[5]));
//                      CurrentInputInFloat = ADC_K_Ix*canmsg.Icell[0] ;
//                      xCell[0].fCellCurrent = (-ADC_K_Ix*(canmsg.Icell[0]) +
    canmsg.FloatOffset);
//                  break;
//             }
//            // Now let's make sure we estimate the SOC
//            if (SOC_Estimate == 0)
//            {
//                vBmsEstInit( xCell, 0 );
//                SOC_Estimate++;
////                Pack_IQ.CellSOC_CAN[0] = _IQ15(xCell[ 0 ].fCellSoc);
////                Pack_IQ.CellSOC_CAN_V2[0] = _IQ14(xCell[ 0 ].fCellSoc);
////                SOCMessages.txMsgData[0] = 1; // This is SOC information
////                SOCMessages.txMsgData[1] = 0;// This is the module average
////                SOCMessages.txMsgData[2] = (uint16_t)Pack_IQ.CellSOC_CAN[0]>>8;
     // This is the first cell in this Module
////                SOCMessages.txMsgData[3] = (uint16_t)Pack_IQ.CellSOC_CAN[0];//
    This is the first cell in this Module
////                SOCMessages.txMsgData[4] = 0; // This is the second cell in
    this Module
////                SOCMessages.txMsgData[5] = 0; // This is the second cell in
    this Module
////                SOCMessages.txMsgData[6] = 0; // This is the third cell in this
     Module
////                SOCMessages.txMsgData[7] = 0; // This is the third cell in this
     Module
////                CAN_sendMessage(CANA_BASE, 14, MSG_DATA_LENGTH, SOCMessages.
    txMsgData); // We are sending from mailbox #14
//            }
//            else
```

```
//          {
//              // Step in time
//              vBmsEstStep( xCell, 0);
//              // Transmit the estimated SOC
//          }
//    }
////    else if(canmsg.status == ComputerDebug_Mailbox_13) // Computer Debug
////    {
////        CAN_readMessage(CANA_BASE, ComputerDebug_Mailbox_13, canmsg.rxMsgData)
    ;
////    }
//    else   // Unexpected message received
//    {
//    }
//    Pack_IQ.CellSOC_CAN[0] = _IQ15(xCell[ 0 ].fCellSoc);
//    Pack_IQ.CellSOC_CAN_V2[0] = _IQ14(xCell[ 0 ].fCellSoc);
//    SOCMessages.txMsgData[0] = 1; // This is SOC information
//    SOCMessages.txMsgData[1] = 0;// This is the module average
//    SOCMessages.txMsgData[2] = (uint16_t)Pack_IQ.CellSOC_CAN[0]>>8; // This is
    the first cell in this Module
//    SOCMessages.txMsgData[3] = (uint16_t)Pack_IQ.CellSOC_CAN[0];// This is the
    first cell in this Module
//    SOCMessages.txMsgData[4] = 0; // This is the second cell in this Module
//    SOCMessages.txMsgData[5] = 0; // This is the second cell in this Module
//    SOCMessages.txMsgData[6] = 0; // This is the third cell in this Module
//    SOCMessages.txMsgData[7] = 0; // This is the third cell in this Module
//    CAN_sendMessage(CANA_BASE, 14, MSG_DATA_LENGTH, SOCMessages.txMsgData); //
    We are sending from mailbox #14
//    DebugCount++;
//       CAN_clearInterruptStatus(CANA_BASE, canmsg.status); // Clear the
    interrupt from message object
//
```

```
//      CAN_clearGlobalInterruptStatus(CANA_BASE, CAN_GLOBAL_INT_CANINT0); //
    Clear the global interrupt flag for the CAN interrupt line
//
//      Interrupt_clearACKGroup(INTERRUPT_ACK_GROUP9);  // Acknowledge this
    interrupt located in group 9
    canmsg.status = CAN_getInterruptCause(CANA_BASE); // Read the CAN-A interrupt
        status to find the cause of the interrupt
    if(canmsg.status >= Module_Mailbox_1 && canmsg.status <= Module_Mailbox_10) //
         This is a message from the module we are looking for
    {


           CAN_readMessage(CANA_BASE, canmsg.status, canmsg.rxMsgData);
            MATLABmsg[0] = canmsg.rxMsgData[0];
            MATLABmsg[1] = canmsg.rxMsgData[1];
            MATLABmsg[2] = canmsg.rxMsgData[2];
            MATLABmsg[3] = canmsg.rxMsgData[3];
            MATLABmsg[4] = canmsg.rxMsgData[4];
            MATLABmsg[5] = canmsg.rxMsgData[5];
            MATLABmsg[6] = canmsg.rxMsgData[6];
            MATLABmsg[7] = canmsg.rxMsgData[7];

           IALL = _IQ8toF(_IQ8((MATLABmsg[1]<<8)>>8) + (_IQ8(MATLABmsg[2])>>8));
           Vdcref = _IQ8toF(_IQ8((MATLABmsg[3]<<8)>>8) + (_IQ8(MATLABmsg[4])>>8));

           startupbits = MATLABmsg[7];
           SOC_Estimate = (startupbits>>5) & 0x0001;
           DroopEn = (startupbits>>6) & 0x0001;

    }

    // Transmit values to MATLAB over CAN
```

```
uint16_t ModuleIndex;

for( ModuleIndex=0; ModuleIndex<4; ModuleIndex++)
{
    Pack_IQ.CellSOC_CAN[ModuleIndex] = _IQ15(xCell[ ModuleIndex ].fCellSoc);
    SOCMessages.txMsgData[2*ModuleIndex] = (uint16_t)Pack_IQ.CellSOC_CAN[
        ModuleIndex]>>8; // This is the ModuleIndex cell in this Module
    SOCMessages.txMsgData[2*ModuleIndex+1] = (uint16_t)Pack_IQ.CellSOC_CAN[
        ModuleIndex];// This is the ModuleIndex cell in this Module
}
CAN_sendMessage(CANA_BASE, CAN_Mailbox_SOC, MSG_DATA_LENGTH, SOCMessages.
    txMsgData); // We are sending from mailbox #14

for( ModuleIndex=0; ModuleIndex<4; ModuleIndex++)
{
    Pack_IQ.CellVoltage_CAN[ModuleIndex] = _IQ12(xCell[ ModuleIndex ].
        fCellVoltage);
    VoltageMessages.txMsgData[2*ModuleIndex] = (uint16_t)Pack_IQ.
        CellVoltage_CAN[ModuleIndex]>>8; // This is the ModuleIndex cell in
        this Module
    VoltageMessages.txMsgData[2*ModuleIndex+1] = (uint16_t)Pack_IQ.
        CellVoltage_CAN[ModuleIndex];// This is the ModuleIndex cell in this
        Module
}
CAN_sendMessage(CANA_BASE, CAN_Mailbox_Voltage, MSG_DATA_LENGTH,
    VoltageMessages.txMsgData); // We are sending from mailbox #15

for( ModuleIndex=0; ModuleIndex<4; ModuleIndex++)
{
    Pack_IQ.CellCurrent_CAN[ModuleIndex] = _IQ8(xCell[ ModuleIndex ].
        fCellCurrent);
```

```
        CurrentMessages.txMsgData[2*ModuleIndex] = (uint16_t)Pack_IQ.
            CellCurrent_CAN[ModuleIndex]>>8; // This is the ModuleIndex cell in
            this Module
        CurrentMessages.txMsgData[2*ModuleIndex+1] = (uint16_t)Pack_IQ.
            CellCurrent_CAN[ModuleIndex];// This is the ModuleIndex cell in this
            Module
    }
    CAN_sendMessage(CANA_BASE, CAN_Mailbox_Current, MSG_DATA_LENGTH,
        CurrentMessages.txMsgData); // We are sending from mailbox #16

////////////////////////////////////////////////////////////////////////////////////////////////

    //Data from module 2
    for( ModuleIndex=4; ModuleIndex<8; ModuleIndex++)
    {
        Pack_IQ.CellSOC_CAN[ModuleIndex] = _IQ15(xCell[ ModuleIndex ].fCellSoc);
        SOCMessages2.txMsgData[2*(ModuleIndex-4)] = (uint16_t)Pack_IQ.CellSOC_CAN[
            ModuleIndex]>>8; // This is the ModuleIndex cell in this Module
        SOCMessages2.txMsgData[2*(ModuleIndex-4)+1] = (uint16_t)Pack_IQ.
            CellSOC_CAN[ModuleIndex];// This is the ModuleIndex cell in this
            Module
    }
    CAN_sendMessage(CANA_BASE, CAN_Mailbox_SOC2, MSG_DATA_LENGTH, SOCMessages2.
        txMsgData); // We are sending from mailbox #17

    for( ModuleIndex=4; ModuleIndex<8; ModuleIndex++)
    {
        Pack_IQ.CellVoltage_CAN[ModuleIndex] = _IQ12(xCell[ ModuleIndex ].
            fCellVoltage);
        VoltageMessages2.txMsgData[2*(ModuleIndex-4)] = (uint16_t)Pack_IQ.
            CellVoltage_CAN[ModuleIndex]>>8; // This is the ModuleIndex cell in
            this Module
```

```
    VoltageMessages2.txMsgData[2*(ModuleIndex-4)+1] = (uint16_t)Pack_IQ.
        CellVoltage_CAN[ModuleIndex];// This is the ModuleIndex cell in this
        Module
}
CAN_sendMessage(CANA_BASE, CAN_Mailbox_Voltage2, MSG_DATA_LENGTH,
    VoltageMessages2.txMsgData); // We are sending from mailbox #18

for( ModuleIndex=4; ModuleIndex<8; ModuleIndex++)
{
    Pack_IQ.CellCurrent_CAN[ModuleIndex] = _IQ8(xCell[ ModuleIndex ].
        fCellCurrent);
    CurrentMessages2.txMsgData[2*(ModuleIndex-4)] = (uint16_t)Pack_IQ.
        CellCurrent_CAN[ModuleIndex]>>8; // This is the ModuleIndex cell in
        this Module
    CurrentMessages2.txMsgData[2*(ModuleIndex-4)+1] = (uint16_t)Pack_IQ.
        CellCurrent_CAN[ModuleIndex];// This is the ModuleIndex cell in this
        Module
}
CAN_sendMessage(CANA_BASE, CAN_Mailbox_Current2, MSG_DATA_LENGTH,
    CurrentMessages2.txMsgData); // We are sending from mailbox #19



      DebugCount++;
   CAN_clearInterruptStatus(CANA_BASE, canmsg.status); // Clear the interrupt
       from message object

   CAN_clearGlobalInterruptStatus(CANA_BASE, CAN_GLOBAL_INT_CANINT0); // Clear
       the global interrupt flag for the CAN interrupt line

   Interrupt_clearACKGroup(INTERRUPT_ACK_GROUP9);  // Acknowledge this
       interrupt located in group 9
```

```
}

/***************************************************************************
*  FUNCTION      : void initADCSOC()
*     return     : void
*     arg        : void
*  Created by    : Josh Wooten
*  Date created  : 02/22/2021
*  Description   : Initialize the ADC start-of-conversion
*  Notes         :
***************************************************************************/
void initADCSOC()
{
    // Initialize for c adc
    // Select the channels to convert and the end of conversion flag
    //
    EALLOW;

    AdccRegs.ADCSOC0CTL.bit.CHSEL = 1; // SOC0 will convert pin C1
                                       // 0:C0 1:C1  2:C2  3:C3
                                       // 4:C4  5:C5   6:C6   7:C7
                                       // 8:C8  9:C9   A:C10  B:C11
                                       // C:C12 D:C13  E:C14  F:C15
    AdccRegs.ADCSOC0CTL.bit.ACQPS = 9; // Sample window is 10 SYSCLK cycles.
        Number plus 1
    AdccRegs.ADCSOC0CTL.bit.TRIGSEL = 5; // Trigger on ePWM1 SOCA

    AdccRegs.ADCINTSEL1N2.bit.INT1SEL = 0; // End of SOC0 will set INT1 flag
    AdccRegs.ADCINTSEL1N2.bit.INT1E = 1; // Enable INT1 flag
    AdccRegs.ADCINTFLGCLR.bit.ADCINT1 = 1; // Make sure INT1 flag is cleared
```

```
    EDIS;
}

/*******************************************************************************
*  FUNCTION      : void initADC()
*     return     : void
*     arg        : void
*  Created by    : Josh Wooten
*  Date created  : 02/22/2021
*  Description   : Initialize the ADC
*  Notes         :
*******************************************************************************/
void initADC(void) //Not used
{
    //
        // Setup VREF as internal
        //
     //  SetVREF(ADC_ADCC, ADC_INTERNAL, ADC_VREF3P3);

        ADC_setVREF(ADC_ADCC, ADC_INTERNAL, ADC_VREF2P5);

        EALLOW;

        //
        // Set ADCCLK divider to /4
        //
        AdccRegs.ADCCTL2.bit.PRESCALE = 6;

        //
        // Set pulse positions to late
        //
        AdccRegs.ADCCTL1.bit.INTPULSEPOS = 1;
```

```
        //
        // Power up the ADC and then delay for 1 ms
        //
        AdccRegs.ADCCTL1.bit.ADCPWDNZ = 1;
        EDIS;

        DELAY_US(1000);
}
/******************************************************************************
*  FUNCTION      : void initEPWM()
*     return     : void
*     arg        : void
*  Created by    : Josh Wooten
*  Date created  : 02/22/2021
*  Description   : Initialize the Pulse width modules
*  Notes         :
******************************************************************************/
void initEPWM(void)
{
    EALLOW;

        EPwm1Regs.ETSEL.bit.SOCAEN = 0;  // Disable SOC on A group
        EPwm1Regs.ETSEL.bit.SOCASEL = 4; // Select SOC on up-count
        EPwm1Regs.ETPS.bit.SOCAPRD = 1;  // Generate pulse on 1st event

        EPwm1Regs.CMPA.bit.CMPA = 0x0800; // Set compare A value to 2048 counts
        EPwm1Regs.TBPRD = 0xC000;        // Set period to 4096*12 counts

        EPwm1Regs.TBCTL.bit.CTRMODE = 3; // Freeze counter

        EDIS;
```

```
}
/**************************************************************************
* FUNCTION      : void ChangeDroopValuesOverCAN(void)
*     return    : void
*     arg       : void
* Created by    : Mohamed Ahmed
* Date created  : 07/14/2018
* Description   : Detect Series or parallel connection
* Notes         :
**************************************************************************/
//void ChangeDroopValuesOverCAN(uint16_t DroopValue)
//{
//  if (ChangeDroopOnce)
//  {
//
//      TransmitToModules[0] = 3;
//      TransmitToModules[1] = DroopValue;
//      TransmitToModules[2] = 1; // Set to 0 to ignore the calibrated offset!
//      TransmitToModules[3] = 0;
//      TransmitToModules[4] = 0;
//      TransmitToModules[5] = 0;
//      TransmitToModules[6] = 0;
//      TransmitToModules[7] = 0;
//
//      CAN_sendMessage(CANB_BASE, 15, MSG_DATA_LENGTH, TransmitToModules); //
    Send Delta Vref and Irefs to the appropriate Module
//      CAN_sendMessage(CANB_BASE, 16, MSG_DATA_LENGTH, TransmitToModules); //
    Send Delta Vref and Irefs to the appropriate Module
//      CAN_sendMessage(CANB_BASE, 17, MSG_DATA_LENGTH, TransmitToModules); //
    Send Delta Vref and Irefs to the appropriate Module
//      CAN_sendMessage(CANB_BASE, 18, MSG_DATA_LENGTH, TransmitToModules); //
    Send Delta Vref and Irefs to the appropriate Module
```

```
//      CAN_sendMessage(CANB_BASE, 19, MSG_DATA_LENGTH, TransmitToModules); //
    Send Delta Vref and Irefs to the appropriate Module
//      CAN_sendMessage(CANB_BASE, 20, MSG_DATA_LENGTH, TransmitToModules); //
    Send Delta Vref and Irefs to the appropriate Module
//      CAN_sendMessage(CANB_BASE, 21, MSG_DATA_LENGTH, TransmitToModules); //
    Send Delta Vref and Irefs to the appropriate Module
//      CAN_sendMessage(CANB_BASE, 22, MSG_DATA_LENGTH, TransmitToModules); //
    Send Delta Vref and Irefs to the appropriate Module
//      CAN_sendMessage(CANB_BASE, 23, MSG_DATA_LENGTH, TransmitToModules); //
    Send Delta Vref and Irefs to the appropriate Module
//      CAN_sendMessage(CANB_BASE, 24, MSG_DATA_LENGTH, TransmitToModules); //
    Send Delta Vref and Irefs to the appropriate Module
//      ChangeDroopOnce = 0;
//  }
//
//}


/*****************************************************************************
 *  FUNCTION      : ONeSecondSOCEstimator(void)
 *     return     : void
 *     arg        : void
 *  Created by    : MOhamed
 *  Date created  : 10/23/2018
 *  Description   : SOC Estimation every one second
 *  Notes         :
 *****************************************************************************/
void ONeSecondSOCEstimator(void)
{
//   uint16_t CellIndex = 0;
//   uint16_t ModuleIndex = 0;
//
```

```
//    for(ModuleIndex = 0; ModuleIndex< 10; ModuleIndex++)
//    {
//        for(CellIndex = 3*ModuleIndex; CellIndex< (3*ModuleIndex + 3); CellIndex
    ++)
//            {
//                vBmsEstStep( xCell, CellIndex);
//            }
//    }
 //   vBmsEstStep( xCell, CellIndex);
}
/*****************************************************************************
  *  FUNCTION       : SOCInitialization(void)
  *     return      : void
  *     arg         : void
  *  Created by     : MOhamed
  *  Date created   : 10/23/2018
  *  Description    : SOC Initialization Function
  *  Notes          :
  *****************************************************************************/
void SOCInitialization(void)
{

    // Get initial SOC's
//    uint16_t i = 0;
//    uint16_t j = 0;
//    for(j = 0; j< 10; j++) // Make sure that modules have sent data to ensure
    that useful data is put into initial SOC estimation
//    {
//        for(i = 3*j; i< (3*j+3); i++)
//        {
//           vBmsEstInit( xCell, i ); // Initial Calculation of SOC for each cell i
//        }
```

```
//   }
 //  vBmsEstInit( xCell, i );

}


/*****************************************************************************
  * FUNCTION      : OneSecondTemperatureCalculator(void)
  *     return    : void
  *     arg       : void
  * Created by    : MOhamed
  * Date created  : 10/23/2018
  * Description   : Temperature Interpolation every one second
  * Notes         :
  *****************************************************************************/
void OneSecondTemperatureCalculator(void)
{

//   uint16_t CellIndex = 0;
//   uint16_t ModuleIndex = 0;
//
//   for(ModuleIndex = 0; ModuleIndex< 10; ModuleIndex++)
//   {
//       PowerBoardThermistor(&boardTemp, ModuleIndex);
//       // Update the temperature value for each cell in the Handle
//       for(CellIndex = 3*ModuleIndex; CellIndex< (3*ModuleIndex+3); CellIndex++)
//       {
//           xCell[CellIndex].fCellTemperature = boardTemp.Result[ModuleIndex];;
    // Initial Calculation of SOC for each cell i
//       }
//   }
}
```

```
/***************************************************************************
 * FUNCTION      : OneSecondHandle(void)
 *     return    : void
 *     arg       : void
 * Created by    : MOhamed
 * Date created  : 10/23/2018
 * Description   : One Second Function Call
 * Notes         :
 ***************************************************************************/
void OneSecondHandle(void)
{

    if (SOC_Estimate == 0)
    {
 //      SOCInitialization();
        //AllowRetrieve = 1;
        //CanQ0.InitializeI2CData = 2; // Here, retrieve all the data from I2C
            EEPROM
    }
    else
    {
        //if (AllowRetrieve == 2)
        //{
            // Estimate SOC
 //          ONeSecondSOCEstimator();
        //}
    }


    // Calculate Deltas and SOCs and then transmit them over CAN
    //BMSAveraging();
```

```
//   uint16_t MOduleIndex = 0;
//    for(MOduleIndex=0; MOduleIndex<10; MOduleIndex++)
//    {
//        SOCMessages.txMsgData[0] = 1; // This is SOC information
//        SOCMessages.txMsgData[1] = 0;// This is the module average
//        SOCMessages.txMsgData[2] = (uint16_t)Pack_IQ.CellSOC_CAN[3*MOduleIndex
    ]>>8; // This is the first cell in this Module
//        SOCMessages.txMsgData[3] = (uint16_t)Pack_IQ.CellSOC_CAN[3*MOduleIndex
    ];// This is the first cell in this Module
//        SOCMessages.txMsgData[4] = (uint16_t)Pack_IQ.CellSOC_CAN[3*MOduleIndex +
     1]>>8; // This is the second cell in this Module
//        SOCMessages.txMsgData[5] = (uint16_t)Pack_IQ.CellSOC_CAN[3*MOduleIndex +
     1]; // This is the second cell in this Module
//        SOCMessages.txMsgData[6] = (uint16_t)Pack_IQ.CellSOC_CAN[3*MOduleIndex +
     2]>>8; // This is the third cell in this Module
//        SOCMessages.txMsgData[7] = (uint16_t)Pack_IQ.CellSOC_CAN[3*MOduleIndex +
     2]; // This is the third cell in this Module
//        CAN_sendMessage(CANA_BASE, MOduleIndex+14, MSG_DATA_LENGTH, SOCMessages.
    txMsgData);
//    }

// Modified by Josh
//
//
//
//        SOCMessages.txMsgData[0] = a; // This is a test varriable
//        SOCMessages.txMsgData[1] = 0;
//        SOCMessages.txMsgData[2] = 0;
//        SOCMessages.txMsgData[3] = 0;
//        SOCMessages.txMsgData[4] = 0;
```

```
//        SOCMessages.txMsgData[5] = 0;
//        SOCMessages.txMsgData[6] = 0;
//        SOCMessages.txMsgData[7] = 0;
//        CAN_sendMessage(CANB_BASE, 13, MSG_DATA_LENGTH, SOCMessages.txMsgData);


}
```