THE DEVELOPMENT OF AN INTEGRATED BATTERY MANAGEMENT SYSTEM AND CHARGER

A Thesis

Presented to

The Graduate Faculty of the University of Akron

In Partial Fulfillment

of the Requirements for the Degree

Master of Science

Thomas V. Vo

August, 2014

THE DEVELOPMENT OF AN INTEGRATED BATTERY MANAGEMENT SYSTEM AND CHARGER

Thomas V. Vo

Thesis

Approved

_____
Advisor
Dr. Tom T. Hartley

_____
Committee Member
Dr. Seungdeog Choi

_____
Committee Member
Dr. Yilmaz Sozer

Accepted

_____
Department Chair
Dr. Abbas Omar

_____
Dean of the College
Dr. George K. Haritos

_____
Dean of the Graduate School
Dr. George R. Newkome

_____
Date

ABSTRACT

An investigation into battery management for lithium-based battery packs was performed. Out of the investigation of the various management/balancing methodologies came a proposed management methodology that is integrated with a charging system and utilizes cost-effective, lossy, bypass resistors for cell balancing. This integration allows the management system to cater the charging current to the needs of the battery pack and overcome the limitations of the lossy bypass on its own. To first investigate this concept, a $LiFePO_4$ cell model was obtained. This was done using a cell discharging procedure and characterization process that provides a mathematical first-principles cell model. The obtained model was then used to simulate various pack configurations, battery management configurations, including the proposed management method. The results from these simulations demonstrated that the proposed management methodology balanced cell voltages within a battery pack in as little as a single charge cycle. To confirm this concept a manually hand-controlled experiment, consisting of voltmeter monitoring cell voltages, manual activation of lossy bypass resistors, and manual adjustments of charging current, was performed. The results from this experiment confirmed the ability to balance the cell voltages within a single cycle. Hardware and software was developed to automate the proposed management methodology. Data collected from the automated implementation was in agreement with the performed simulations and successfully demonstrated a

functional automated version of the proposed integrated battery management system and charger.

ACKNOWLEDGMENTS

I would like to thank my advisor Dr. Tom T. Hartley for sharing his methods, knowledge, his patience, and believing in me. Graduate school and life in general for that matter, is a non-linear system, hard to model and even harder yet to control. I'm grateful for having such a great role model during my undergraduate and graduate years to help optimize my output from this system.

Thanks to Matt Taschner for helping me along the journey as well. We learned, developed, discussed many things together and the experience would not have been the same without him. He pushed through getting his thesis done and was an inspiration for me to do the same.

A big thanks goes out to Erik Rinaldo for all of the practical knowledge he passed on in an endless number of projects/tasks he always was willing to help me with. Thanks to both Erik Rinaldo and Greg Lewis for the support in ordering the endless amount of items we had purchased for all research purposes. Also a huge thanks goes out to Gay Boden for helping me push through the massive amount of logistics, forms, and paper-work to be able to complete my thesis and work.

Last and not least, I would like to thank my family and friends for their unending support as I pursue my interests. Without a great support network this whole process would

have been all the more challenging on my own. No matter the contribution, large or small, thank you all for your kind words of support and help.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

Table Page

CHAPTER I

INTRODUCTION

Portable battery applications today can consist of either a single electrochemical cell (which will further be referred to simply as a "cell") or a battery pack that consists of a combination of two or more cells. However, in industry, the term battery is often interchangeably used to describe both an individual cell as well as a battery pack. A cell that is designed such that it only is discharged once, and not recharged, is referred to as a primary cell. A cell that is designed such that it can be discharged and recharged multiple times is known as a secondary cell. It is the secondary cells and their applications that are the focus of this research. When using these secondary cells, special care must be taken to ensure the cells' voltage, temperature, and maximum current remains within the manufacturer's specified operating range. Maintaining this range will help to ensure that the manufacturer's specified cell capacity, the total possible amount of charge before damage occurs, stays consistent with each charge and discharge cycle. This number of charge and discharge cycles is also referred to as the cells' cycle life. Cell manufacturer's specifications often provide a minimum cycle life to which the cell should be capable of maintaining a specified storage capacity. That is, provided there are not manufacturing defects and the cells are properly used and maintained. In general, this minimum cell cycle

life can increase or decrease with more aggressive or lower demanding deviations from the cell manufacturer's recommended voltage operating ranges (Hartmann, 2008).

1.1 Types of Cell Combinations (Battery Packs)

In applications where a single cell is utilized, monitoring and maintaining the recommended manufacturer's operating range is straightforward. This is because the charging device and load can be controlled to cater to that individual cell's operation. On the other hand, in applications where there is a battery consisting of multiple cells, ensuring each individual cell stays inside the required voltage range can become more challenging. Battery packs can consist of two cells (i.e. handheld portable consumer products) to several thousand (i.e. electric vehicle applications). Battery packs can be constructed such that they are connected in series, parallel, or a combination of series and parallel. The various combinations of cells and the challenges of monitoring and managing them will be discussed in the following paragraphs.

1.1.1 Series-Connected Cells

In applications that demand higher voltages than an individual cell can provide, a number of cells can be connected in series to create a battery pack with the required higher voltage. This type of battery pack configuration is referred to as series-connected or simply as a series string. During operation of this battery configuration type, the current during charging and discharging is the same current through each cell within the string. This is advantageous when measuring current as only a single current measurement is required. However, when it comes to monitoring cell voltages, circuitry is required for each individual cell. In addition, due to the fact that a single current charges and discharges these cells, how well balanced the stored energy in each cell is becomes an issue. Depending on

the balance of energy within the pack and individual cell construction variations, cells may finish charging and discharging at different times. Variations between the cells is what causes the individual cell voltages to go outside the manufacturer's specified range, and cause damage to individual cells. As will be discussed later in this document, several methods exist that allow the cells in the string to be balanced.

In general, it is possible for one or more cells to fail in an open-circuit state or in a short-circuit state. If a cell fails in an open-circuit condition, there is no longer a current path. This renders the series-connected battery pack unusable. On the other hand, a short-circuit cell failure condition, while not ideal, will still allow for current to be drawn from the battery pack. The short-circuit condition will reduce the voltage of the overall battery pack by the voltage that was once provided from the short-circuited cell.

1.1.2 Parallel-Connected Cells

In applications that demand higher current than an individual cell can provide, multiple cells may be connected in parallel to provide higher current draw. Since the cells are connected in parallel, all of the cells' voltages are equal. If there is voltage difference before the cells are connected in parallel, the higher potential cells will transfer energy to the lower potential cells until the voltages balance (to prevent a large surge of current the cells should be relatively balanced). In terms of monitoring circuitry, paralleling cells is beneficial in that only one circuit is needed to measure the whole parallel group's voltage. However, the exact individual currents provided by each cell are unknown unless the individual cell currents are measured. In terms of cell failures in this configuration, unlike the series configuration, the open-circuit fault condition does not render this pack configuration useless. However, with this type of failure the application will draw higher

currents from the remaining cells. If the remaining parallel cells cannot handle the new current demand, they could also be damaged. In the case of the short-circuit fault condition, this will cause all of the other parallel connected cells to discharge through the faulted cell and render the battery pack un-useable.

1.1.3 Series and Parallel Connected Cells

In applications that require both higher voltages and higher current, a combination of series-connected and parallel-connected cells is required to form the battery pack. There are two common combinations that exist. The first combination is paralleled-series-connected cell strings. This combination consists of several cells connected to form series strings, which in turn are connected in parallel. The second combination is "series-connected parallel groupings". This combination consists of several cells connected to form parallel cell groupings. These parallel cell groupings are in turn connected in series. Based upon the previous discussion, the configuration that stands out in terms of reliability is the first combination of paralleled-series-connected cell strings. In this configuration, the fault that rendered the original independent series string unusable, has less of an impact and improves overall reliability (McDowall, 2005). This is because an open-circuit failure results in only one-of-the-many strings to become unusable. The remaining strings can still provide energy to allow the application to continue. In the second configuration, the fault that renders the series-connected-parallel groupings unusable still poses an issue. If a cell fails in a short-circuit condition, it will still drain energy from the remaining cells within that entire grouping.

1.2 Goals of Research

The focus of this research is to monitor and protect cells within a battery pack that are connected in a series-connected configuration. To ensure that the cells stay balanced within a series-connected configuration, the use of a cost-effective, lossy, cell bypass is investigated. Although only a single series-string is investigated, the principles of this research can be carried over to paralleled-series-string battery packs. In terms of cell chemistry, lithium-based chemistries are of focus in this research due to their high energy density, lower costs, and the traction it is gaining within the market place (Krieger, 2103) (Vincent, 2000). To perform this research a first-principles cell model is obtained via data collected through a given procedure. This model obtained in turn is then used to simulate the cell balancing circuitry and its effectiveness. In order to further improve the utility of the cost-effective, lossy, cell-bypass, a concept of integrating the cell balancing system and the charging device is proposed and investigated.

1.3 Thesis Outline

The research performed is provided over seven chapters. Chapter I is an introduction to cells and batteries. Chapter II provides a history of cells as well as background information on cells, cell balancing methods, and a proposed balancing method and charging system is provided. Chapter III provides details on a procedure that is performed to collect data used to determine a cell model. Chapter IV uses the obtained model to demonstrate the effectiveness of a passive bypass for balancing a series-connected battery pack. Also simulated in Chapter IV is the proposed balancing system. Chapter V and Chapter VI provides a summary of the software and hardware, respectively, used to implement an interactive charger and passive bypass balancing battery management

system. Chapter VII provides results from experiments performed to test the proposed method. Chapter VIII concludes the thesis along with recommendations for future work in this area.

CHAPTER II

BACKGROUND AND RELATED WORK

This chapter discusses the history of electrochemical cells and battery packs. A description of the inner working of electrochemical cells is given along with the various cell chemistries. This section ends with operational considerations when using electrochemical cells, a summary on charging techniques, and a summary on cell-balancing techniques.

2.1 The History of Electrochemical Cells

Although the earliest electric cells were in existence over 2,000 years ago, the history of the battery truly begins in the 18th and 19th centuries. People like Alessandro Volta (1745 – 1827) and Luigi Galvani (1737 – 1798) carried out the groundwork that led to an electrochemical energy storage device, and their names live on in terms such as "volts" and the "galvanic cell" (Bergveld, 2001).

Around the 1800s the voltaic column, the world's first working battery was created. This battery consisted of alternate copper and zinc plates, separated by scraps of cloth saturated with acid. Using this first battery, Volta discovered that certain liquids initiated chemical reactions between metals, thereby generating electrical energy (Bergveld, 2001).

In the early 19th century, Volta worked in close collaboration with the French National Institute to further develop the battery. The first battery suitable for mass production was developed in 1802 by the chemist Dr. William Cruickshank (Anders, 2003). He stacked alternating layers of copper and zinc sheets of equal dimensions. This stack was then placed in a sealed wooden chest and sealed with cement. This chest in turn was then filled with a saline lye.

In 1859 the French physicist Gaston Planté used conductor plates in dilute sulphuric acid, which led to the first rechargeable battery (Bergveld, 2001). Previous batteries developed up to this point were all primary cells and were not re-charged. Planté's invention led to the first rechargeable secondary battery, which was a lead-acid chemistry that is used to this day.

Over the following decades the lead battery underwent a number of significant further developments. By introducing a variety of alloys, battery performance was greatly improved which then minimized the need for maintenance. Experimentation with lithium batteries began in 1912 under G.N. Lewis, and in the 1970s the first non-rechargeable lithium batteries were sold. A research team managed by Akira Yoshino of Asahi Chemical, Japan built the first lithium ion battery prototype in 1985, a rechargeable and more stable version of the lithium battery; followed by Sony that commercialized the lithium ion battery in 1991 (Vincent, 2000) (The Economist, 2008). The next section will discuss the inner workings of these cells.

2.2 Electrochemical cells

Electrochemical cells are aptly named because they can provide electrical energy stored in the form of chemical energy. This energy can then be utilized at a later time by conversion of the chemical energy back to electrical energy. At a high level, a cell is comprised of two metal electrodes suspended in an electrolyte. At the interfaces between the electrodes and electrolyte are where chemical reactions take place allowing for the conversion to and from electrical energy (depending on whether the cell is being used as a load or if it is being charged, respectively). The two electrodes switch roles as anodes or cathodes, depending on whether they are being charged or discharged. The two electrodes will be designated as PE and NE, for the positive electrode and negative electrode, respectively. In the case of charging, the PE and NE are designated as the anode and cathode, respectively. During charging, at the interface between the electrolyte and the PE, an oxidation reaction occurs. As a result of this, electrons are free to flow out to the external charging circuitry. Simultaneously, at the electrolyte and the NE interface, a chemical reduction process occurs that accepts electrons from the external charge circuitry. Inside the electrolyte, negatively charged ions produced from the reduction reaction occurring at the NE, move towards the PE. Similarly within the electrolyte, positively charged ions produced by the oxidation reaction at the PE move towards the NE. Thus, the flow of current is possible because of the electrons, which travel in and out of the cell, use the electrodes as a medium to travel through. The availability of free electrons in the PE and the "room" to accept electrons in the NE is made possible through the chemical reactions made at the electrolyte and electrode interface. These reactions can continue because the positive and negative ions use the electrolyte as a medium to travel through.

The discharge process through a cell is the reverse of the charging process. By definition, the PE and NE switch roles as the anode and cathode. The location of the reactions and direction of movements for the electrons and ions are also switched. This can be more clearly seen by comparing Figure 2.1 and Figure 2.2. Pictured within in the figure, in between the PE and the NE, is an electrically isolating separator. The separator is typically a porous material that still allows the ions to easily travel through, but not so porous that it does not behave well as an electrical insulator (Kumar et al., 2010) (Bergveld, 2001).

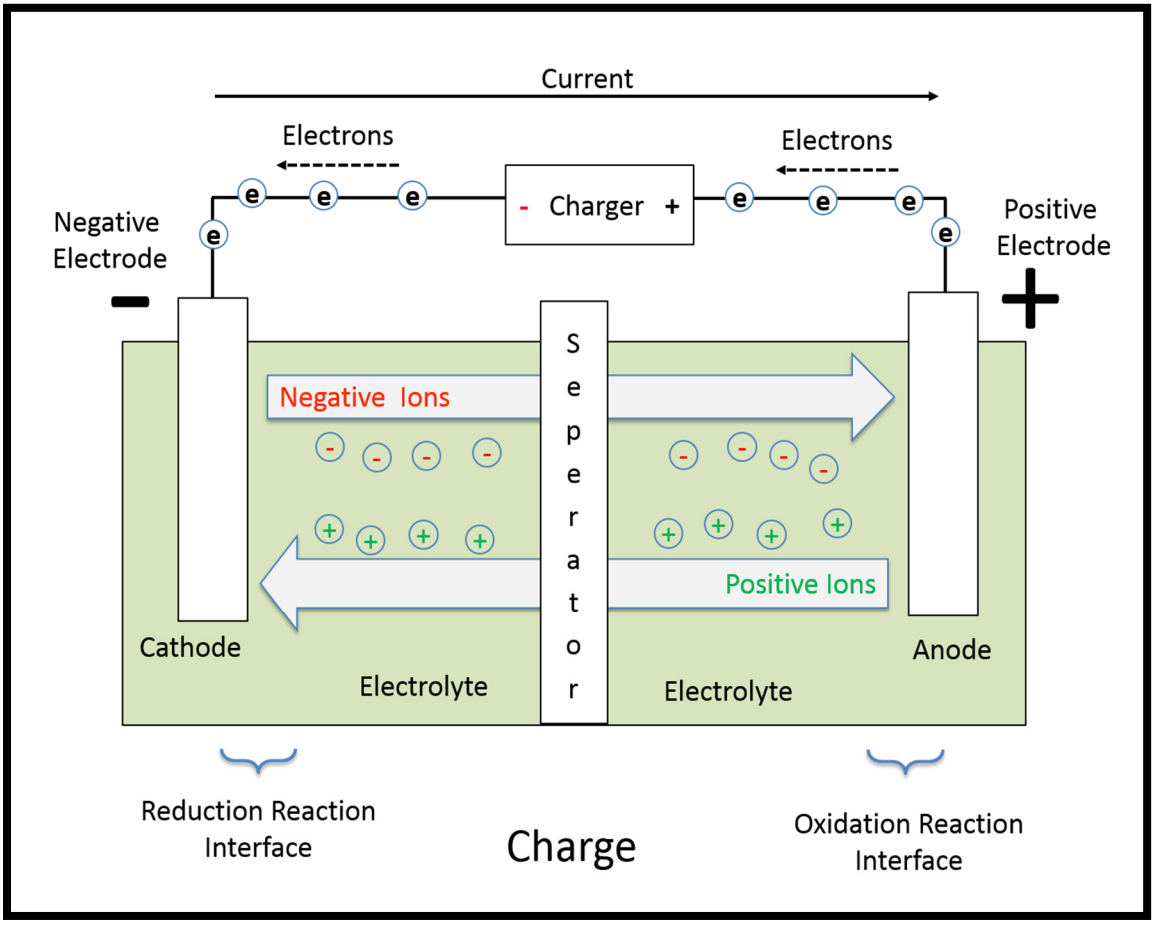

Figure 2.1: Electrochemical Cell pictured during charge process and electrons and ion travel paths.

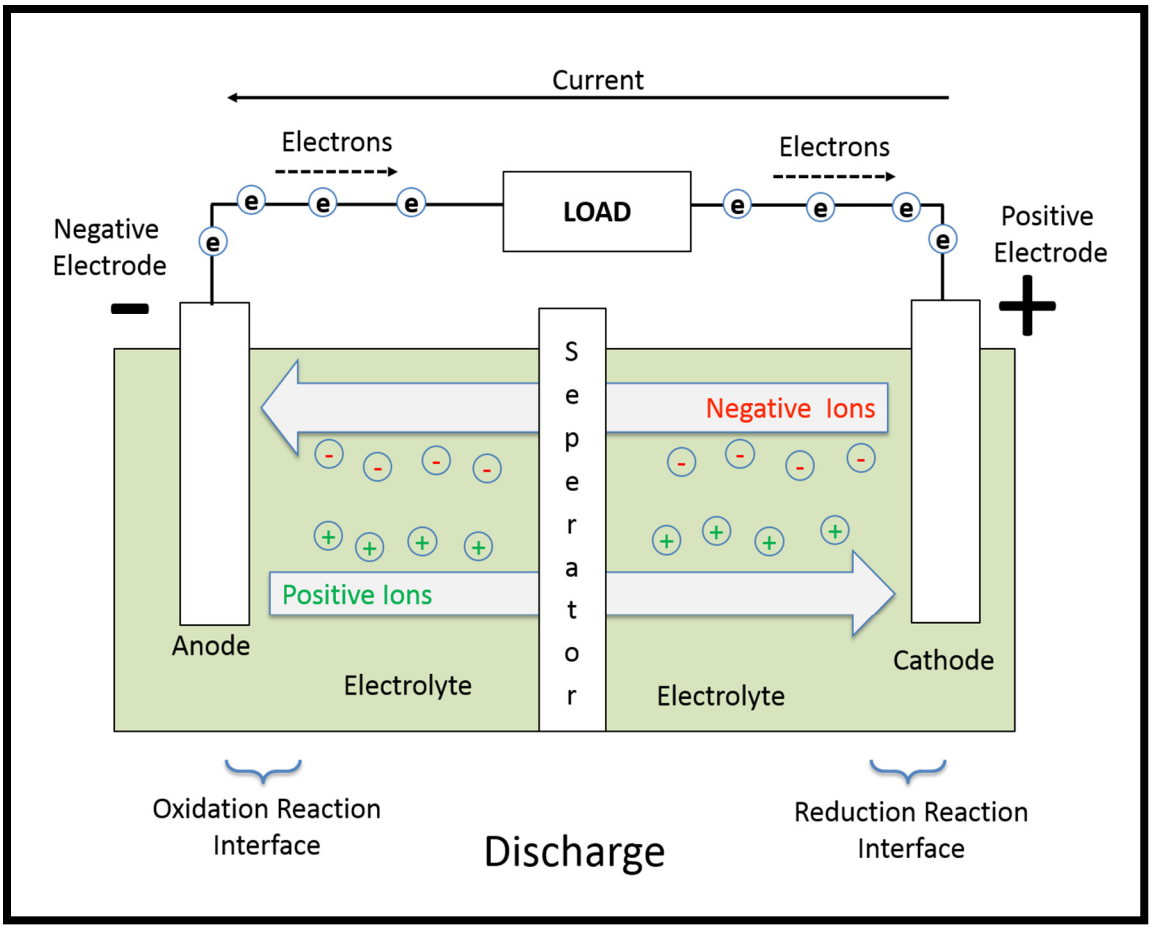

Figure 2.2: Electrochemical Cell pictured during discharge process and electrons and ion travel paths.

2.3 Cell Chemistries

Although the focus of this research is on rechargeable Lithium Ion (Li-Ion) battery cells, a brief overview of other battery chemistries is given in the following sections. For reference, a comparison of multiple cell chemistries and storage technologies are given in the Ragone Plot in Figure 2.3 (Kalhammer, 2007). As can be seen, the various types of lithium-based energy storage devices, outperform the other storage devices in terms of energy storage and instantaneous power per kilogram of material.

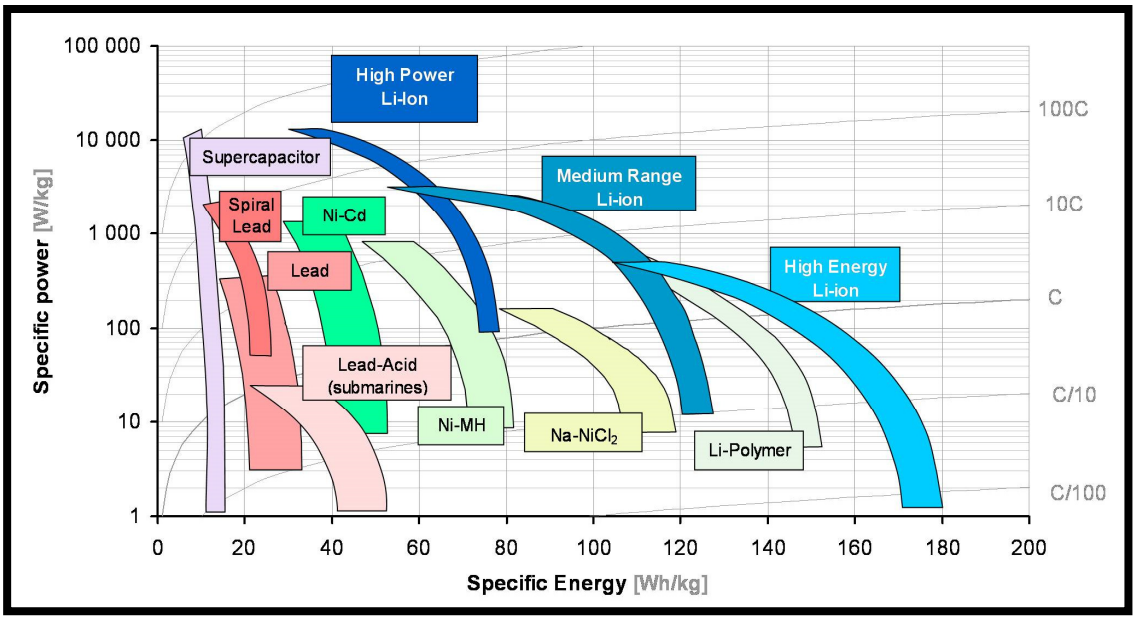

Figure 2.3: Ragone plot comparing several types of cell chemistries and other energy storage devices (Kalhammer, 2007).

2.3.1 Lead-Acid

The oldest form of rechargeable battery is the lead-acid battery. The lead-acid battery was developed into two main designations. The first being sealed lead-acid (SLA) and the second being large valve regulated lead-acid (VRLA). Both battery types should not be operated at high depths of discharge to preserve cycle life. An advantage of the lead-acid battery is that cells may be balanced by charging the battery pack with a slow, trickle charge for long durations. Typical charge times are 8-16 hours, and typical cycle life is 200-300 cycles. This short cycle life is due to the corrosion that occurs for the positive electrode which causes depletion of the active material and expansion of the plates. The optimum operating temperature for lead-acid is 25 degrees C, and as a general rule, a change of 8 degrees C will cut the cycle life in half. The lead-acid battery has the lowest

energy density of rechargeable cells, making it unsuitable for smaller consumer electronics (Buchmann, 2011).

2.3.2 Nickel-Cadmium

The Nickel-Cadmium (NiCd) battery has advantages in that it is well-suited for fast charging and rigorous discharge conditions. NiCd batteries have a typical cycle life of 1,000 cycles, and offer good performance at low temperatures without significant impact to cycle life. The NiCd has a decent shelf life as compared with other rechargeable cells, and is the lowest-cost battery in terms of cost per cycle. Unfortunately, the NiCd battery suffers from the memory effect, and relatively low energy density. The memory effect occurs when NiCd cells are charged after being only partially discharged several times. This effect causes the cell to lose some of its prior capacity and it "remembers" the new smaller capacity (Buchmann, 2011). To prevent this, the NiCd cell simply has to be discharged to its full rated capacity.

2.3.3 Nickel-Metal Hydride

The Nickel-Metal Hydride (NiMH) battery offers 30-40 percent higher energy density as compared to NiCd, and is less prone to the memory effect. The disadvantages of the NiMH are related to its performance which is depleted if cycled at high discharge rates. Additionally, the NiMH requires a more complex charging algorithm, and more frequent maintenance to prevent crystalline formation. This higher-cost battery has a high self-discharge rate (50% higher than NiCd), and is sensitive to high temperatures.

2.3.4 Lithium Based Chemistries

The Lithium-Ion (Li-Ion) battery is probably the most well-known chemistry on the market today. Original development for this battery began in 1912, but was not commercially available until 1991 due to concerns of safety. The energy density of Li-ion cells are twice that of NiCd, while also offering a low self-discharge rate and low-maintenance with a high cycle life. Because of safety concerns, the Li-ion cells require protective circuits (such as those discussed in this thesis), and are still relatively expensive to manufacture. The cells support higher voltages, and also offer moderate discharge currents. There are several types of lithium-based cell chemistries as will be discussed in the following sections (Buchmann, 2011).

2.3.4.1 Lithium Cobalt Oxide ($LiCoO_2$)

This chemistry is used in small portable electronics such as cameras, phones, tablets, and some laptops. The positive electrode is derived from cobalt oxide and the negative electrode is graphite carbon. Typical charge and discharge current maximum's are relatively small. Anything higher causes overall cell life to decrease. While the specific energy, the cells' ability to store energy per unit of weight, of this chemistry is high, the specific power is low. The specific power is a measure of how much instantaneous power the cell can provide in a short period of time (Buchmann, 2011).

2.3.4.2 Lithium Manganese Oxide ($LiMnO_2$)

This chemistry was first introduced in 1996 and had a positive electrode made of lithium manganese oxide. Due to the architecture, a three-dimensional spinel structure, ions can flow more freely, decreasing internal resistance. This allows for high currents which make this chemistry suitable for applications such as power tools, medical equipment, and

electric vehicles. Compared to lithium cobalt, lithium manganese has a capacity roughly one-third the size of lithium cobalt's storage capacity (Buchmann, 2011).

2.3.4.3 Lithium Iron Phosphate ($LiFePO_4$)

In 1996, researchers at the University of Texas implemented phosphate as a positive electrode for a lithium based cell. This material which can be designed with nano-scale phosphate which offers low resistance, high current rating, and long cycle life. Along with the aforementioned benefits and a relatively higher abuse tolerance, this technology finds applications also in power tools, electric vehicles, and back-up power applications. In comparison to the $LiMnO_2$ chemistry, due to $LiFePO_4$'s lower 3.2-3.3V nominal voltage, it has a slightly lower specific energy (Buchmann, 2011) (Padhi et. al, 1996).

2.3.4.4 Lithium-Titanate (Li_2TiO_3)

This chemistry has been around since the 1980s. In this cell chemistry Lithium-titanate, as opposed to carbon, is used for the negative electrode. This also forms a spinel structure as with the $LiMnO_2$ cell chemistry, enabling lower internal resistance. This allows this chemistry to safely charge and discharge quickly at high currents, and has low-temperature discharge characteristics (down to -30^{o}C). Due to the fact that a single cell provides 2.4V nominally, its specific energy is relatively low (Buchmann, 2011).

2.4 Charging

There are several cell charging methods that can be used when it comes to cell charging. A few of these modes are constant-current charging, constant-voltage charging, trickle-charging, float-charging, and pulse-charging. Since the focus of this research is on lithium-based cell technologies, the commonly used constant-current, constant-voltage

method will be discussed in greater detail. The other methods will be briefly discussed in the following sections.

2.4.1 Lithium-based Battery Charging

A lithium-based battery charger is typically a fixed voltage source that is current limited. The charger will charge the battery at a constant current (CC) until the battery voltage reaches a pre-set value, the set-point voltage. This charging current is typically limited to a 1C charge rate, however, higher or lower charge rates may be possible depending on the recommendation of the manufacturer. The "C rate" is a relative specification based upon the Ah rating of the cell. This is easily explained in the form of an example. If a cell has a rated capacity of 2Ah, 1C, 2C, and 3C corresponds with charge (or discharge) currents of 2 amps, 4 amps, and 6 amps. About 65% of the total charge is delivered to the battery during the constant current phase of charging.

Once the set-point voltage is reached during the CC phase, the charger will begin to decrease current to maintain the set-point voltage on the battery. This is known as the constant voltage (CV) phase of charging. Most major Li-ion cell manufacturers recommend 4.200V +/- 50 mV as the ideal set point voltage. The constant-voltage cut-off time is typically given by the manufacturer with a desired degree of accuracy to ensure the specified cell cycle-life and capacity can be met. It is important to note that this value is specified for the voltage available at the cell's terminals. When current is actively charged or discharged from the cell, the series resistance in the form of terminal connections and/or wiring provides measurement offsets from the true cell's terminal voltage. A method to overcome this may be accomplished by providing two high impedance measurement wires connected directly to the cell's terminals. Since these two wires are high impedance and

are not part of the cell's main current path, the offset voltage measurement issues are overcome (Bergveld, 2001).

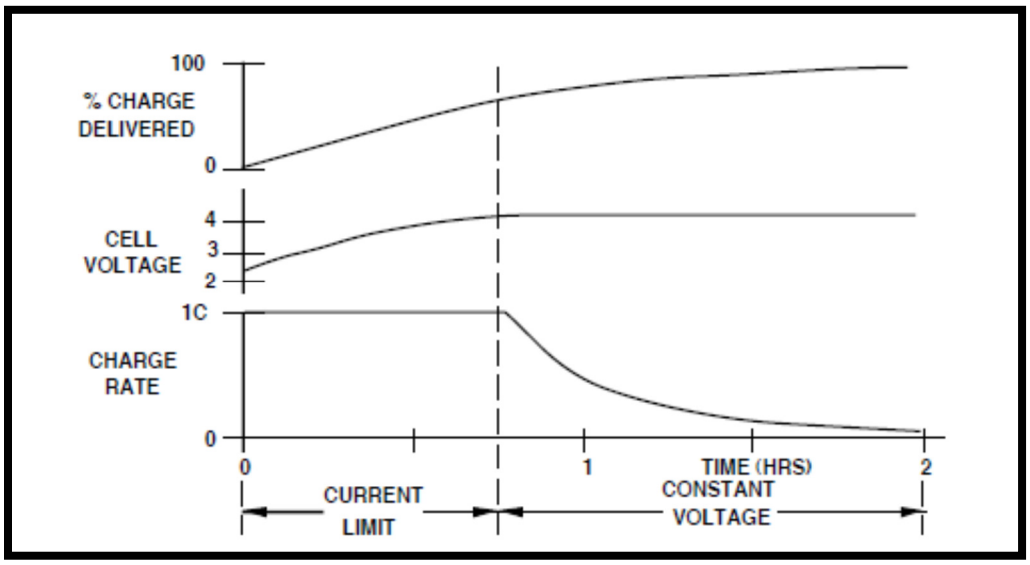

Figure 2.4: A CC/CV charger first applies constant current rate until the battery reaches a set-point voltage, charging current is then reduced to maintain the set-point voltage (Simpson, 2011).

2.4.2 Floating/Trickle Charging

When charging battery packs with chemistries of lead-acid, NiCd, or NiMH, float charging and/or trickle charging is performed. This type of charging is utilized after a charge cycle has already completed, and while the battery or cell is unused. The charge current is typically small (less than 0.05 C) and is used to both compensate for the self-discharge of the battery pack as well as balance the pack at the end of charge. This is important in the case of lead-acid especially as sulfation occurs more readily at lower discharge states, leading to poor battery performance.

In terms of lithium based chemistries, holding this chemistry at higher voltages for long durations instead leads to lower cycle life. Some possible causes attributing to the lowering of cycle life are electrolyte decomposition, formation of a passivation layer on the negative electrode, and dendrite formation. Each of these conditions can decrease cycle life as the available active materials required for charging/discharging decreases. In the extreme case, the dendrites can grow so that a short is formed between the positive and negative electrodes (Bergveld, 2001).

2.4.3 Lithium Ion Battery Discharging

The end-of-discharge voltage for a Li-ion cell is typically 2.5V on average. At this point, approximately 95% of the energy is depleted, and from there, the cell voltage drops rapidly if discharging were allowed to continue. To protect the cell from overdischarging, most battery management systems will prevent operation beyond this low-voltage cutoff. When a load is removed from a battery after discharge, the voltage will gradually recover by a small amount.

In terms of choice of discharge rates and overall cell cycle life, higher discharge rates lead to faster capacity fade as can be seen in Figure 2.5 (Ning et al, 2003).

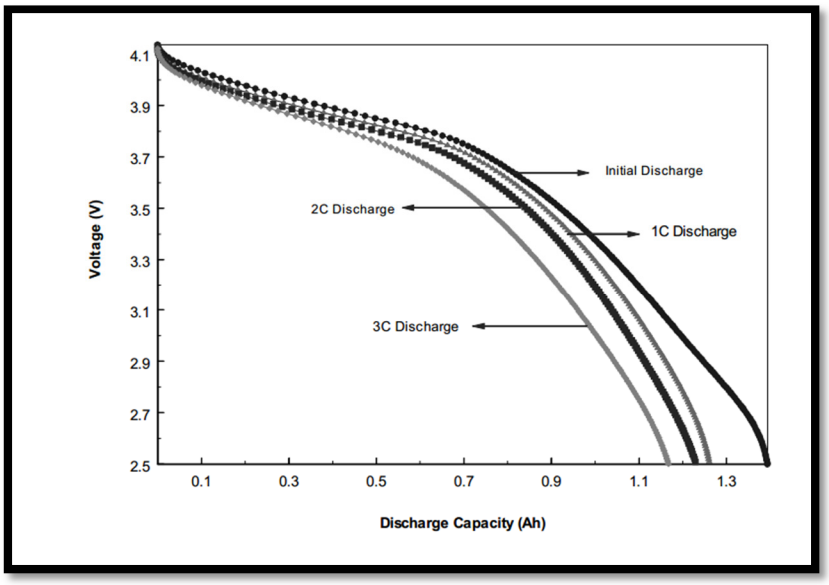

Figure 2.5: Initial capacity of $LiCoO_2$ cells before and after 300 cycles at varying discharge currents. The capacity fade increases with higher c-rates (Ning et al, 2003).

Historically, Li-ion cells had been considered unsuitable for high current loads. In recent years, however, many Li-ion systems permit discharge rates upwards of 50C. This means that a cell rated at 1.5Ah can provide a steady load of 45 amps, and this is being achieved primarily by lowering the internal resistance through optimizing the surface area between the active cell materials (Choi et. al, 2002).

2.5 Need for Cell Balancing and Battery Management

Chargers for series-connected packs often look at the battery pack's overall voltage for controlling the charging current. If each cell is identical and evenly charged, it could be assumed that each cell is at the same voltage. However, in practice individual cells' state-of-charge (SoC), or the amount of stored energy within in a cell, is likely to be unequal. During charging, this can cause cells with higher SoC to reach higher than recommended

operating voltages. Similarly, during discharge, cells with lower SoC will reach lower than recommended operating voltages.

The choice of what voltage to charge a cell to and what voltage to discharge a cell to ultimately affect the number of charge/discharge cycles that can be achieved with that cell (Hartmann, 2008). In a series-connected battery pack without a method to balance cell SoC, all SoC's are hard to regulate if the individual cell voltages are very different. Again, this is due to the fact that the same current is used to charge each cell. This will cause one or more cells to become fully-charged faster than others. An option at this point is to simply stop charging the series string and leave the remaining cells not fully charged. However, during discharge, the lowest charged cells will discharge the quickest and then energy in the higher charged cells will go unused. This is non-ideal since the full-capacity of the battery pack is not being utilized.

The main causes of cell imbalance are variations in an individual cell's impedance, capacity, and self-discharge rate. These factors will cause divergence in the cells' voltage over time. Since most battery chargers detect full charge by checking whether the voltage of the entire string of cells has reached the voltage-regulation point, individual cell voltages can vary as long as they do not exceed the limits for overvoltage protection. However, both weak cells (i.e., cells with lower capacity or higher internal impedance) and warm cells tend to exhibit higher voltage than the rest of the series cells at full charge termination. These cells are weakened further by continuous overcharge cycles (Andrea, 2010).

2.6 Cell Balancing Methods Summary

The impact of cell imbalance on run-time performance and battery life in applications using series-connected cells is certainly undesirable. The fundamental solution of cell balancing equalizes the voltages and SoCs among the cells when they are at full charge. Cell balancing is usually categorized into two types, passive and active. With passive balancing, excess energy is dissipated as heat. Active-balancing, on the other hand, is more energy efficient as it attempts to transfer energy conservatively from cell to cell.

The fact that 100% of the excess energy from a higher-energy cell is dissipated as heat makes the passive method less preferable to use during discharge because of the obvious impact on battery run time. Active cell balancing, which utilizes capacitive or inductive charge shuttling to transfer charge between battery cells, is significantly more efficient because energy is transferred to where it is needed instead of being bled off. Of course, the trade-off for this improved efficiency is the need for additional components and complexity at higher cost.

2.6.1 Passive Balancing

The easiest approach to cell balancing is to equalize cell voltages. This can be done by comparing cell voltages with programmable thresholds to determine if cell balancing is needed. If any particular cell hits the threshold, a resistive bypass is enabled. The main disadvantage of this method is the energy lost through the bypassing. This method is known as passive cell balancing (Moore, 2009) (Andrea, 2010). Passive balancing is a cost-effective, lossy, option to reduce the disparity between the SoC of each cell. The bypass resistor allows higher charged cells to burn-off excess energy in the form of heat (Figure 2.6). It also creates a path to allow energy to shunt around a cell in a series-connected

string. The limitation that comes from utilizing a bypass is the amount of heat that is generated from the bypass device itself. One strategy for utilizing a bypass is to simply turn on the bypass device once the cell hits a specific voltage. This allows the rate-of-charge to the highest charged cells to be reduced and allows current to be shunted around that particular cell.

Based upon the charging current and bypass resistance, there are three conditions that can exist when a bypass is implemented. One condition is if the bypass resistor is designed such that it bypasses less than the charging current. This implies that the remaining charging current that is not bypassed continues to charge the cell. The second condition is if the bypass resistor is selected such that it bypasses approximately the same value of the charging current. In this condition, the cell stops charging and the cell's voltage will stay constant. This is because if the cell's voltage goes up, the fixed-resistance bypass will bypass a little more current as well. If the cell's voltage goes down, the fixed-resistance bypass also bypasses less current. The third condition is if the bypass resistor is selected such that it bypasses all of the charging current and also draws additional current from the cell. This allows higher charged cells to decrease their SoC while allowing other cells to increase their SoC.

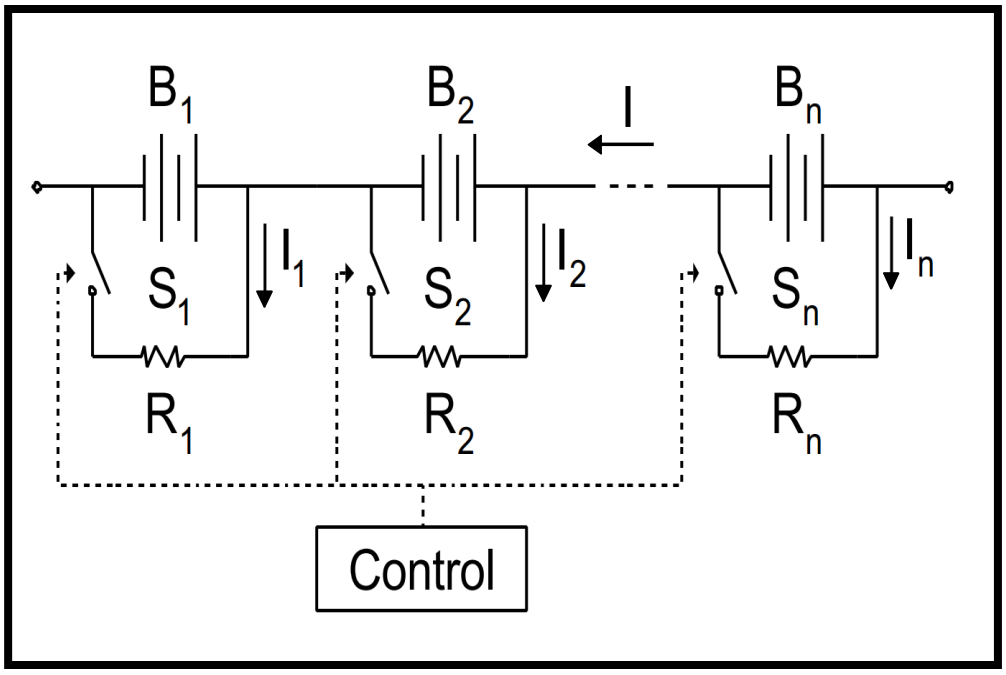

Figure 2.6: Passive Resistive Bypass (Moore, 2009).

2.6.2 Active Balancing

Another approach is active cell balancing. This method overcomes the energy loss of the passive method by using capacitive or inductive charge storage and shuttling to deliver energy to where it is needed most, and with little loss. The disadvantage to this method is in component count, cost, and complexity. Active balancing techniques fall into four main groups (Yevgen, 2009) (Moore, 2001):

1. Cell to cell: energy is transferred between neighboring cells.

2. Cell to battery: energy is removed from cells with highest SoC and dumped to the whole battery.

3. Battery to cell: Energy is removed from the battery pack and transferred to the cells with least SoC.

4. Bidirectional: Based on needs, energy is transferred from cell to battery or vice versa.

These methods may be implemented in a variety of ways. A simple approach to redistribute the energy between the cells is to connect a capacitor first to higher voltage cell, than to lower voltage cell, as shown Figure 2.7.

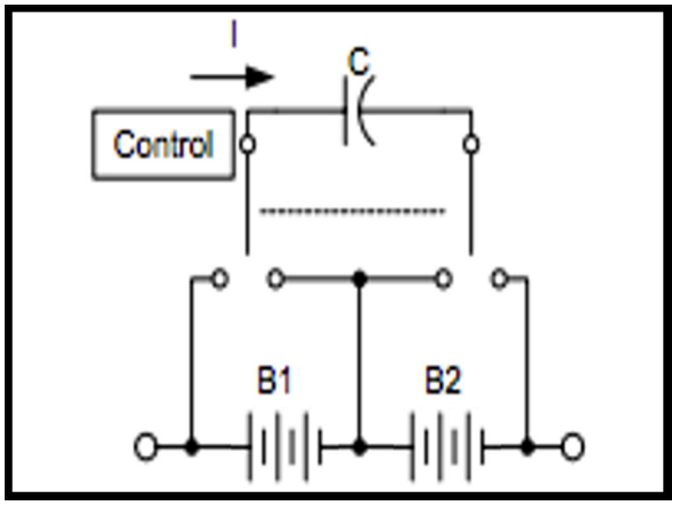

Figure 2.7: Capacitor-Based Shuttle (Yevgen, 2009).

More complicated implementations allow the connection of not only two nearby cells, but also cells for far away in the stack for faster equilibration Figure 2.8.

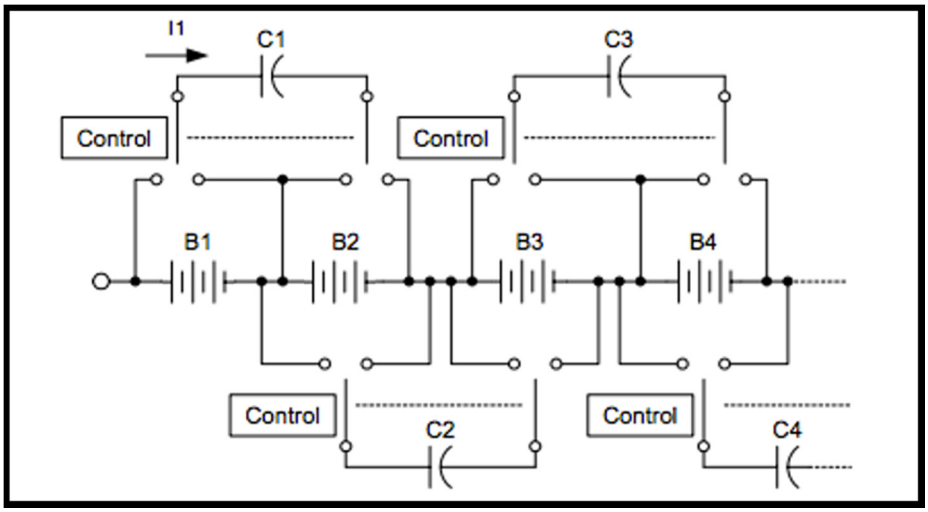

Figure 2.8: Shuttle circuit with remote cells connection capability (Yevgen, 2009).

The main problem with this method is that significant energy losses occur during capacitor charging, due to high currents because of high voltage mismatch. Another problem is that high voltage differences between the unbalanced cells exist only in highly discharged states. Because this method's transfer rate is proportional to cell voltage differences, it only becomes efficient near the end of discharge so that the total amount of unbalance that can be removed during one cycle is low.

Another active balancing method, depicted in Figure 2.9, is implemented by taking energy from the battery pack as a whole and redistributing it to a single cell. This is done by directing the battery pack current through a transformer which is then switched to one of the cells that needs additional charge. However, the efficiency of such a converter is limited, and the need to use a transformer results in increased price and size of the overall solution (Moore, 2001).

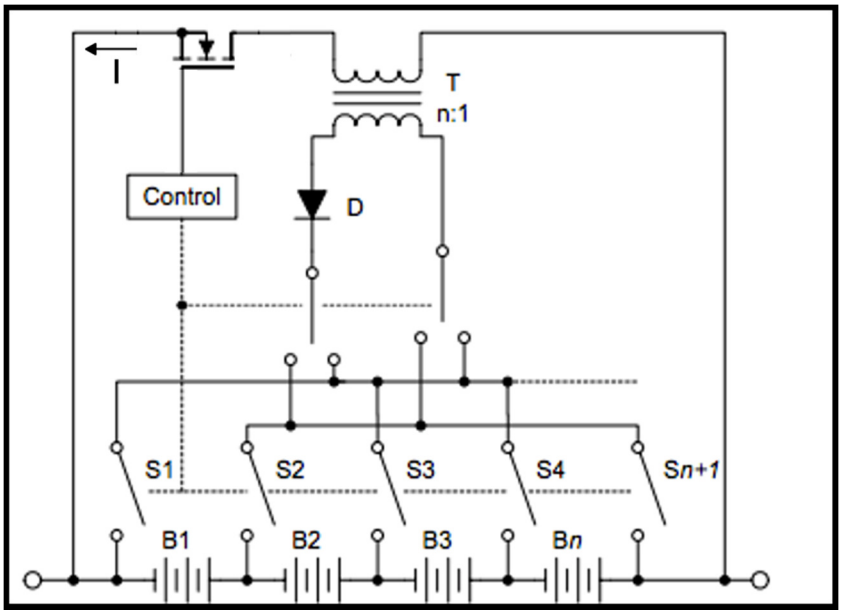

Figure 2.9: Inductive converter cell balancing circuit (Moore, 2001).

2.7 A Proposed Integrated Charger and Battery Management System

A study performed in 2011 compared several balancing methods consisting of both passive and active circuitry (Daowd et al, 2011). In the investigation, each method was evaluated and simulated within MATLAB/Simulink. Each method was graded upon its equalization speed, complexity, size, cost, and efficiency. The method that outperformed others in terms of simplicity, and cost-effectiveness, was the passive resistive shunt bypass. In terms of speed and efficiency, the resistive shunt bypass received average to less-than-average ratings. In the case of many high volume commercial applications, cost and development time is often of great concern. In fact, many available battery management integrated circuit manufacturers, such as Texas Instruments, utilize passive resistive shunts for balancing (Texas Instruments, 2012). In this research, the resistive shunt bypass is used due its popularity, cost-effectiveness, and minimal amount of components. A proposed

method to further improve the speed of balancing with passive resistive shunts and reduce the number of cycles a battery pack stays unbalanced is discussed next.

The proposed battery system will use a BMS that interfaces directly with the charging system as shown in Figure 2.10. The BMS consists of one Individual Cell Management Unit (ICMU) per a cell and a single Battery Pack Master Unit (BPMU) for the entire battery pack. The block diagram of Figure 2.11 depicts an ICMU which consists of circuitry that at a high level, includes a voltage, current and temperature monitor, and an adjustable bypass current device. In terms of the adjustable passive bypass device, this will be achieved by a MOSFET driven in its linear region. In doing so this causes the MOSFET to behave as a voltage-controlled resistor. The adjustability of the bypass current as well as the adjustability of the charging current allows the system to have more flexibility when it comes to equalizing the energy within battery pack's cells.

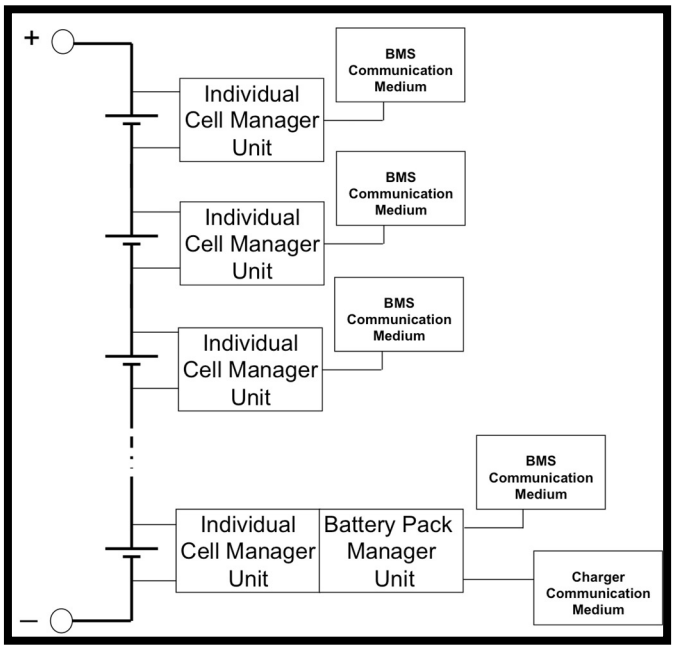

Figure 2.10: High Level Block Diagram for Integrated BMS and Charger System.

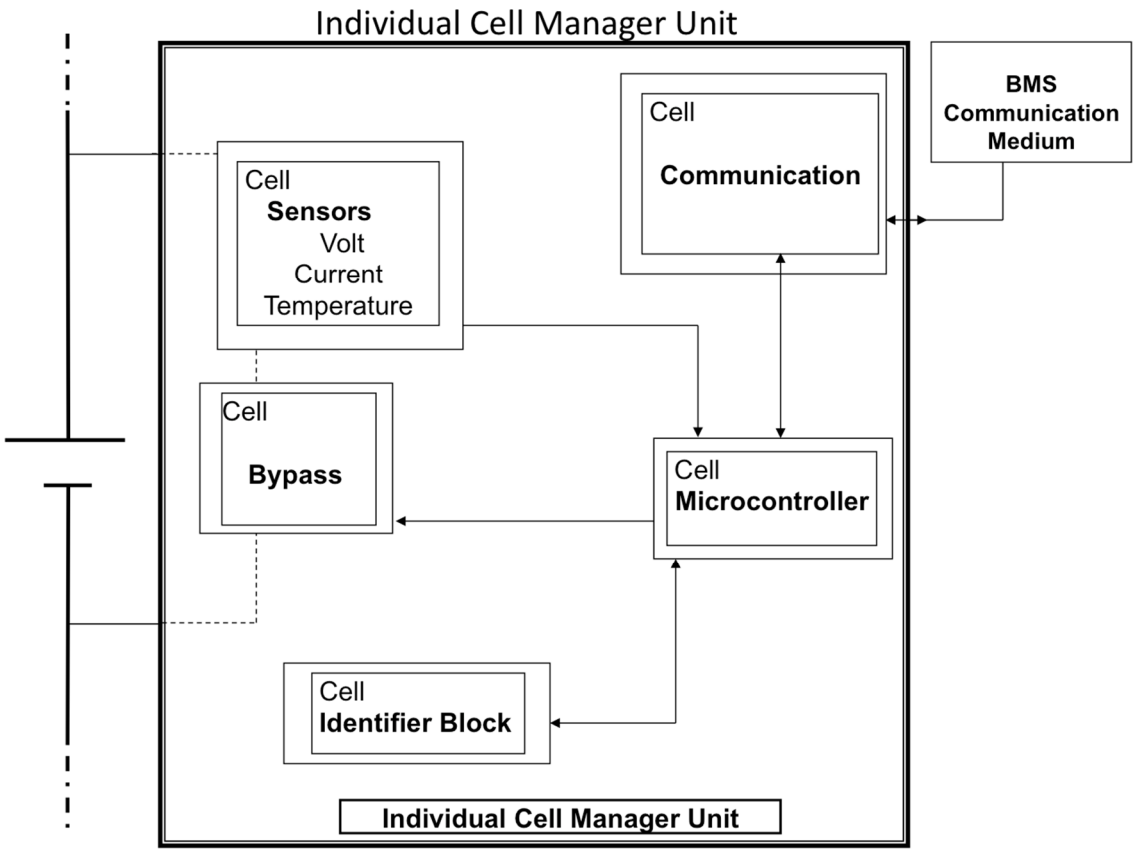

Figure 2.11: Detailed block diagram for Individual Cell Manager Unit within BMS.

The BPMU, shown in Figure 2.12, processes data from each ICMU for monitoring of the battery pack as a whole. The BPMU will communicate battery pack issues during charge and discharge. In addition, the BPMU will also have a communication interface to the charging system. This interface will allow the BPMU to directly control the charging current profile as it demands from the charger. A block diagram of the integrated charger is shown in Figure 2.13.

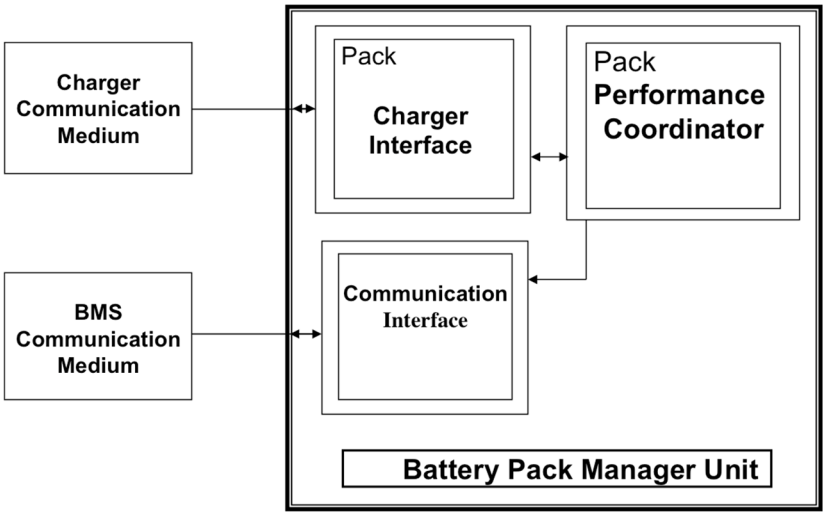

Figure 2.12: Detailed block diagram for Battery Pack Manager Unit within BMS.

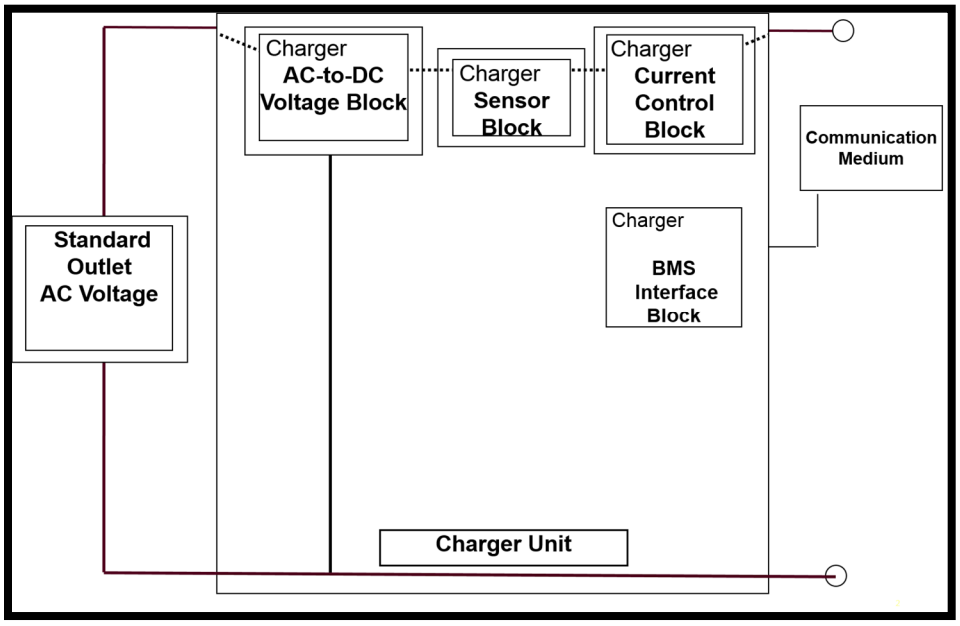

Figure 2.13: Detailed block diagram for Charger for integrated BMS concept.

The sequence of steps for the proposed method of charging and balancing will be discussed next. First, the charging device will initially charge the battery pack at the maximum constant current rate that the battery pack application can accept. When the highest charged cell is nearing its full charge voltage, the BMS can request the charger to

decrease its current as desired. Simultaneously, the BMS can activate bypass current devices on each cell as necessary, shunting as much current as possible around the highest charged cells. In doing so, this allows the remaining lower charged cells to continue charging at a higher rate. If the highest charged cells continue to charge too quickly (even with the bypasses on), the BMS can then request the charging device to lower its charging current as necessary. This continues until the bypass devices can shunt all of the charging current around a given cell and discontinue the charging to any individual cell.

2.8 Summary

This chapter provided a brief history on the discovery and the development of cells. Also provided was background information on cells and how they function electrically and chemically during the charge and discharge processes. Next, an overview of cell chemistries used most in industry was provided. This led to a discussion of methods that can be used to perform cell balancing within a battery pack. Lastly, the chapter concluded with a proposed method for balancing using an integrated charger with a resistive-bypass based battery management system.

CHAPTER III

BATTERY CELL MODELING

This chapter will discuss a procedure that can be used to obtain a mathematical cell model. The benefits of using a cell model and how a cell model may be used for various applications are also discussed.

3.1 Introduction

To be able to simulate the voltage behavior of a cell during usage, either on its own or within a battery pack, an accurate cell model may be used. Using a cell model allows simulations and investigations to be performed of how individual cells behave or multiple cells behave within a battery pack. This allows theories and concepts to be tested without actually spending time on a physical test set-up. This chapter will discuss a cell model and how the parameters for this model are obtained. In order to obtain parameters for the cell model, data must be collected from the cell of interest. The data collected must be obtained following a procedure that allows the cell's dynamics of interest to be observed, as well as either a full charge cycle or a full discharge cycle. After collecting the data, the data is graphed and analyzed to obtain some of the model's parameters. These parameters can vary from cell size, cell chemistry, and even from cell to cell within a manufacturing batch.

With regard to the variability within a manufacturing batch, battery packs designed for space applications go through categorizing newly manufactured cells in order to pick cells with well-matched characteristics. The goal and assumption is that these cells start out with the same SoC and have very similar charge/discharge characteristics as they are used throughout their lifetime. If this is indeed the case, the assumption is that such a battery pack no longer requires a battery management system (Pearson, 2004).

Also, cell modeling improves accuracy estimates for a cell's SoC. This is especially the case in applications where the power profile varies such that the electrochemistry dynamics are not able to reach a state of equilibrium. The cell voltage is dependent on multiple factors such as the charge/discharge current, temperature, and the cell's age. This being the case, it can be difficult to provide an accurate estimate of a cell's SoC solely upon its voltage. Also, a sensor cannot directly be employed to read the cell's SoC directly, however with the use of a mathematic cell model/algorithm, this is possible. An exception for using a cell's voltage as an estimate for its current SoC is when the current (and power) is relatively low such that the open-circuit voltage is close to the running voltage. In cases such as these, it is possible to utilize a table that correlates the SoC with open-circuit voltage (Plett, 2004).

Accuracy of the SoC is important in applications where aggressive power profiles demanded from the battery pack are required. The accuracy also allows more aggressive use of energy available within the battery, as the true SoC of each cell is known. To make this concept clearer, take the following example. In a hypothetical application where the cell's voltage is close to the manufacturer's recommended cut-off voltage, the cell's voltage may not be accurately reflected when a large current is drawn from the cell. Due

to the cell's internal impedance, an internal voltage drop may occur that is correlated with the current draw from the cell. This is an offset from the true cell voltage, and causes an external voltage measurement to appear as if it were below the cut-off voltage. This issue may be alleviated from more accurate SoC estimates via real-time cell models.

3.2 Battery Modeling

The cell model used in this research is derived from a first-principles based structure (Hartley & Jannette, 2005). In other words, the model is based upon established laws of physics. This model is a generalized structure with parameters that are acquired through data collection consisting of voltage, current, and temperature measurements from a cell as it is discharged through a low-resistive load. To model the charge diffusion characteristics of the cell, during the discharge cycle, the load is temporarily removed from the cell in order to observe the resulting transient response. An example of a complete discharge cycle with charge diffusion transient (small transient towards middle of plot) can be observed in Figure 3.1.

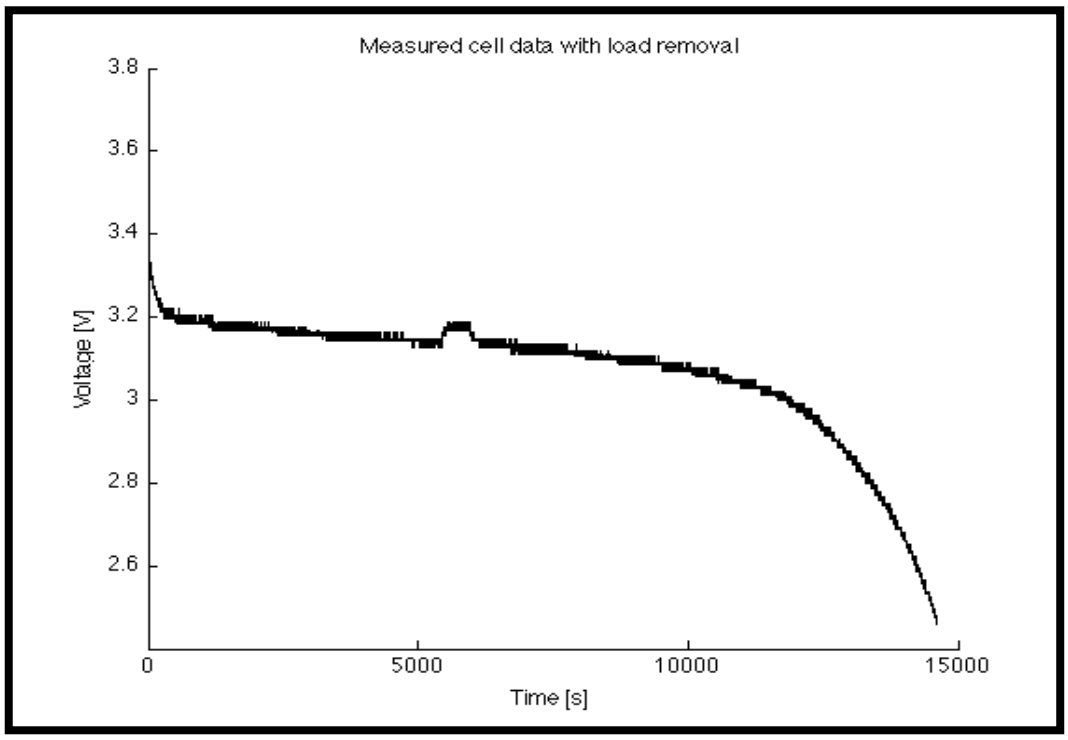

Figure 3.1: Measured Battery Voltage for full cell discharge cycle.

The model obtained will generate a voltage as a function of the stored charge, diffusing charge, and the amount of current flowing in or out of a given cell at any given time, which are denoted by $q_s(t)$, $q_d(t)$, and $i(t)$, respectively.

The state equations of the cell model are the stored-charge rate given by

$$\dot{q}_s(t) = i(t) \tag{1}$$

and the diffusing-charge rate given by

$$\dot{q}_d(t) = \frac{1}{C_d} i(t) - \frac{1}{C_d R_d} q_d(t) \tag{2}$$

where C_d is the diffusion capacitance and R_d is the diffusion resistance (Hartley & Jannette, 2005). The values for the diffusion parameters are obtained by analyzing the diffusion transient response. In order to obtain the stored charge and the diffusing-charge states, equations (1) and (2) are integrated with the measured current during the discharge cycle.

Using only a linear combination of the integrated states $q_s(t), q_d(t)$, and the current, $i(t)$, the linear portion of the voltage curve of Figure 3.1 can be accurately modeled. In order to represent the non-linear portions of the voltage curve in the beginning and end of the discharge cycle, two additional exponential terms dependent on $q_s(t)$ are incorporated. System identification is performed on the data collected using batch least squares to acquire the following parameters

$$\theta = \begin{bmatrix} V_o & R_o & \frac{1}{C_s} & K_d & K_1 & K_2 \end{bmatrix}^T \tag{3}$$

each of which corresponds to an element of

$$P(t) = \begin{bmatrix} 1 & i(t) & q_s(t) & q_d(t) & e^{-A_1 q_s(t)} & e^{A_2 q_s(t)} \end{bmatrix} \tag{4}$$

such that the battery voltage is given by

$$\hat{v}(t) = P(t)\theta \tag{5}$$

$$= (1)(V_o) + (i(t))(R_o) + \left(q_s(t)\right)\left(\frac{1}{C_s}\right)$$

$$+\left(q_d(t)\right)(K_d) + \left(e^{-A_1 q_s(t)}\right)(K_1) + \left(e^{A_2 q_s(t)}\right)(K_2).$$

Based upon the state $q_s(t)$, the SoC is calculated as follows

$$Q_{SoC}(t) = \frac{q_s(t)}{q_{s,max}} 100\% \quad (6)$$

where $q_{s,max}$ is the rated full capacity of a given cell.

The scope of the research did not implement this model in real-time, however, a brief discussion of such an implementation follows. A real-time implementation allows an application to track cell parameters, usually not directly measureable, that provide more information about a cell's condition. For example, the (SoC) can be extracted from the data with higher accuracy than directly measuring the cell's voltage alone (Hartmann, 2008) (Bergveld, 2001). This is especially useful in cases where the cell's current consumption is dynamically changing, which causes voltage measurements alone to be misleading when it is used for SoC estimation. By implementing a cell model on a computing device (e.g. microcontroller, computer, etc...) in real-time, this allows real-time access to information about the aforementioned non-direct measurements.

There are several additional benefits to be gained from a real-time model. Self-discharge can be modeled by measuring a cell's remaining capacity after being stored for a long period of time. With an accurate self-discharge model, it can be determined that a cell is failing by detecting a fast self-discharge rate.

In terms of accuracy of a real-time cell model, it is vital in the case of aggressive designs and high-power usage of battery packs. The ability to accurately predict remaining SoC in a battery pack allows for maximum power draw, and may be estimated without overdischarging a cell or causing it damage. This could also allow a smaller battery pack

be utilized for a given application. This is as opposed to sizing the battery pack with a large design margin and higher cost (Plett, 2004).

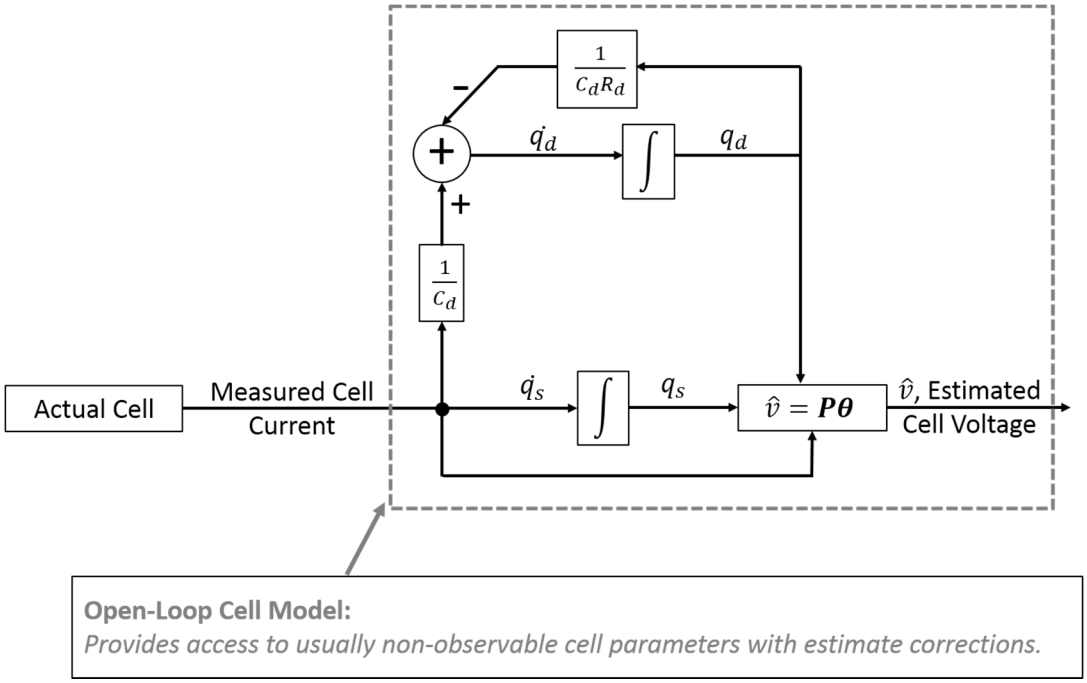

Figure 3.2: Control block diagram of open-loop cell model.

In order to improve accuracy of the model's states, a real-time observer can be implemented to supplement the model running on the computing device. The real-time observer will compare the model's output voltage with an actual measured voltage of the cell of interest by calculating the error,

$$e(t) = v(t) - \hat{v}(t), \tag{7}$$

where $v(t)$ is the cell's measured voltage and $\hat{v}(t)$ is the model's estimated voltage given.

Equations (1) and (2) become

$$\dot{q}_s(t) = i(t) + L_s e(t), \tag{8}$$

$$\dot{q}_d(t) = \frac{1}{C_d} i(t) - \frac{1}{C_d R_d} q_d(t) + L_d e(t), \tag{9}$$

where L_s and L_d are observer gains that are multiplied by the error, $e(t)$, to correct the state estimates, for $q_s(t)$ and $q_d(t)$, respectively, (Plett, 2004) (Vamsi et. al, 2007) (Bergveld, 2001).

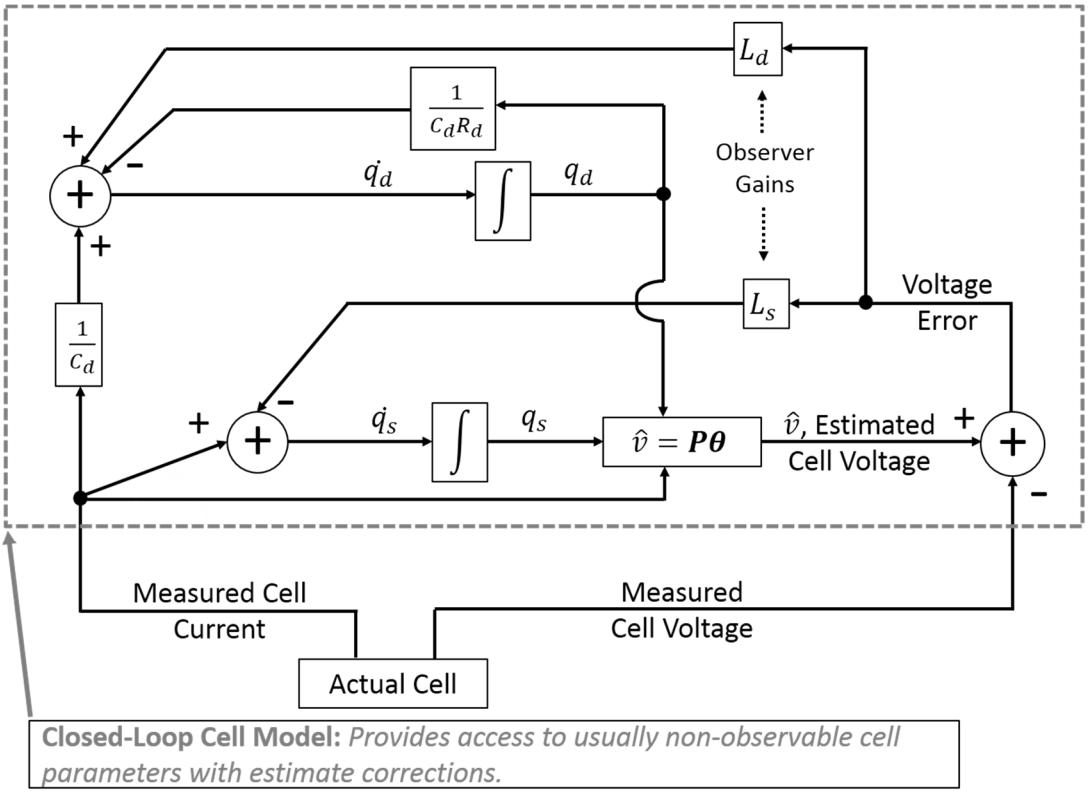

Figure 3.3: Control block diagram of closed-loop cell model with observer.

To further improve estimates, there are additional methods that can be implemented. This includes providing the ability for the cell model's internal parameters to adapt such that it tracks the aging effects of the cell. This is opposed to determining the

cell's parameters from a single set of data and keeping the internal model parameters at fixed values indefinitely. Also, taking operating cell temperatures into the real-time model allows for more accurate cell-model estimates. A possible approach for adapting cell parameters with temperatures is to gather cell operation data at specific temperatures of interest. A polynomial fit may then be used to match temperatures to correlating coefficients (Plett, 2004) (Hartmann, 2008).

3.3 Procedure for Data Collection

In order to get a cell model, data must be collected from the cell during its operation. In this research a fully charged $LiFePO_4$, 3.3V nominal, 60Ah cell was used. The cell was charged per the cell manufacturer's recommendations using a power supply with settings for constant current and constant voltage. The cell was first charged in constant current mode at 20A per the manufacturer's recommended charging current of C/3. The power supply was set for a constant voltage of 4.2V, at which point the charger automatically tapered the charging current to maintain the desired 4.2V. The charge process was stopped when the charging current fell below C/6. The cell was then left to rest several hours to allow the cell's voltage to settle after the charging process.

Next, the cell was discharged via a fixed load resistor bank. The equivalent resistance used during the discharge was ~0.21 ohms which provided a nominal discharge current of ~16A. Both the cell's discharging current and the cell's voltage was logged using a microprocessor development board. These measurements were done via a resistive shunt and voltage divider, respectively. The discharging procedure consisted of connecting the load to the cell, removing the load, re-applying the load, and stopping the discharge process. The step of removing the load was performed in order to capture the dynamics

associated with the cell's charge diffusion. The load was removed long enough until the cell's voltage increased and reached a steady-state. At that point, the load was re-applied until the cell's voltage hit the manufacturer's stated cell cut-off voltage. The data collected in this process is graphed in Figure 3.5.

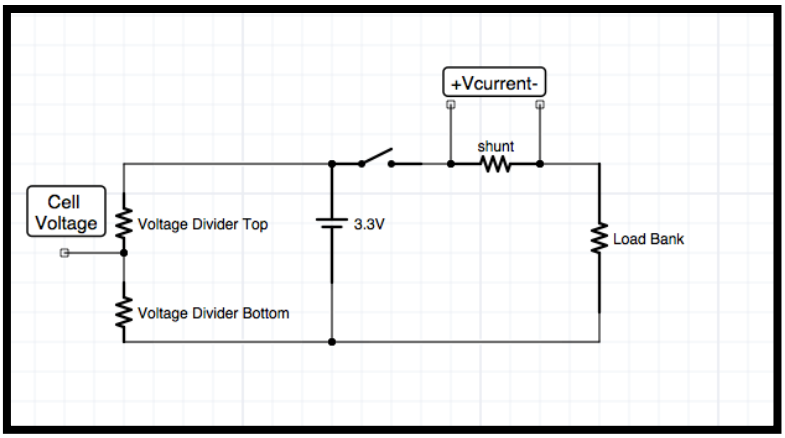

Figure 3.4: Diagram of circuit used to collect cell discharge data.

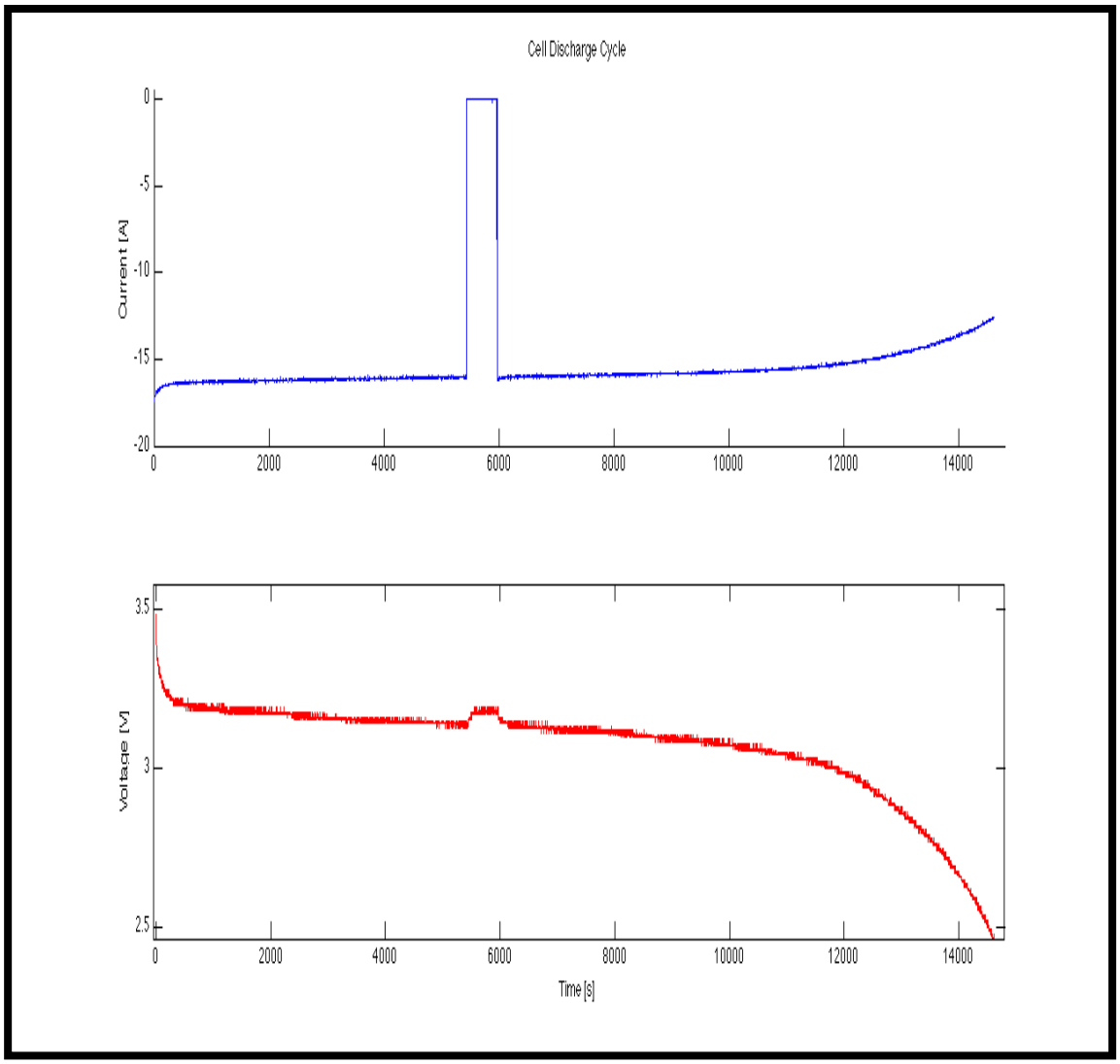

Figure 3.5: Graph of collected discharge data (time axis is in seconds)

3.4 Processing Collected Data

Using the graphed data in Figure 3.5, measurements are made from specific features from the plot. These measurements are then used to calculate and obtain values for the cell model's parameters. The following are the parameters and how they are extracted from the graph.

1. R_o: To determine this value, two other values must be obtained beforehand.
    - The current before load removal (BLR), I_{BLR}, which can visually be determined from the plot.
    - The "instantaneous" change in voltage observed directly after removal of the load. This is designated as the voltage after load removal, $V_{ALR,\Delta}$. In terms of the "instantaneous" change in voltage, what should be observed upon load removal is a sudden jump in voltage due to the lack of a voltage drop across the cell's internal impedance (because of the sudden stop in load current). It is noteworthy to distinguish between the "instantaneous" voltage change and the exponential voltage change due to the cell's diffusion properties (of which the parameters are obtained in the next section), because they occur very closely one after the other in the graph.
    - To calculate R_o the following equation is used: $R_o = \frac{V_{ALR,\Delta}}{I_{BLR}}$
2. Diffusion Time Constant: To determine the Diffusion Time Constant, τ_d , the cell dynamics associated with load removal are analyzed. The following steps are used to determine the time constant:
    - Determine the initial voltage, $V_{Diff,initial}$, immediately after the removal of the load.
    - Next, determine the time, $T_{Diff,initial}$, at which the load was removed.
    - Determine steady-state voltage, $V_{Diff,SS}$, after load removal (this requires that during data collection the cell voltage was allowed to reach a steady-state after the load was removed).

- Calculate the voltage that corresponds with 63% of the voltage difference between the initial voltage and steady-state voltage. This value is the diffusion voltage after one time constant which will be designated as V_{Diff,T_1}, This step is summarized by the following:

$$V_{Diff,T_1} = (V_{Diff,SS} - V_{Diff,initial}) \times 0.63.$$

- Visually determine the time, T_{Diff,T_1}, at which the time at which V_{Diff,T_1} occurs.
- Finally, τ_d can be calculated with:

$$\tau_d = T_{Diff,T_1} - T_{Diff,Initial}$$

3. R_d: To determine $R_d$the following steps can be followed:
   - Visually determine the current, $I_{Diff,initial}$, right before the load is removed.
   - Use $V_{Diff,\tau}$ to determine the current right before the load is removed.
   - R_D may be calculated with:

$$R_d = \frac{V_{Diff,\tau}}{I_{Diff,initial}}$$

4. C_d: may simply be calculated with the obtained values of R_d and τ_d using the following:

$$C_d = \frac{R_d}{\tau_d}$$

5. To determine V_o, two parameters are required, the change in voltage after the load removal and the voltage at the end of the end of the linear region of cell. $V_o = V_{ALR,\Delta}$+ end of linear region voltage.
6. The next step calculates the stored charged and charge diffusion given in Equation (1) and Equation (2), respectively. In order to solve these first order ordinary differential

equations, a numerical method was utilized namely, the Euler method (Hartley et. al, 1994). For each time step during the sampled discharge cycle:

- The stored charge q_s is approximated with the Euler method with a 1s time step ($T_{sample} = 1\ second$). Also for the approximation to be accurate it is important that the initial condition, $q_s[0]$, must be initialized with the estimated SoC of the cell.

  $$q_s[n+1] = q_s[n] + I[n] \times T_{sample}$$

  where T_{sample} is the sampling time of the logged data.

- Likewise, the diffusion charge is approximated using the same numerical method, with the parameters C_d and R_d obtained from steps 2, 3, and 4:

  $$q_d[n+1] = q_d[n] + \left(\frac{I[n]}{C_d} - \frac{q_d[n]}{R_d C_d}\right) \times T_{sample}$$

7. Next, a trial and error method is used to determine A_1 and A_2 of equation 4. These values are chosen based upon the sharpness of the non-linear regions towards the beginning and the end of the discharge curve. This corresponds with the curves at the far left and far right of the voltage plot of Figure 3.5. Step 8 provides a metric which provides feedback on the selection of the values of A_1 and A_2.

8. Next, batch least squares is performed using the collected and calculated data. An array is formed using the parameters from equation 4. Each row in the array corresponds with either the data sampled or calculated values for each time step in the discharge cycle.
    - Theta is solved for in equation 3 using batch least squares
    - $\theta = (P^T P)^{-1} P^T v(t)$
    - Once theta is obtained the Butler-Volmer equation is calculated with the solved parameters and the measured cell current measured during the discharge cycle. This is to grade how well the model predicts the cell voltage.
    - $\hat{v}(t) = P \cdot \theta$
    - The sum of the square of the difference between the model's estimated voltage, $\hat{v}(t)$ and the measured data, $v(t)$, is calculated to determine a metric for the accuracy of the model. This metric can be used for the trial and error selection of the terms in Step 7. The smaller the sum becomes with the choice of A_1 and A_2, the more accurate the model is in predicting the cell voltages.

3.5 Obtained Cell Model Parameters

The values obtained for the model are given in Table 3.1.

Table 3.1: Values for obtained parameters for $LiFePO_4$ cell model.

Parameter	**Value**	**Units**
A_1	22/24000	coloumb^{-1}
A_2	$-10.14/240000$	coloumb^{-1}
R_o	~ 0	ohm
C_d	25052	farad
R_d	0.00243	ohm
τ	61	second
V_o	3.122	volt
$\frac{1}{C_s}$	$4.345\text{E}-7$	farad^{-1}
K_d	1.793	volt
K_1	-1.0696	volt
K_2	$4.21\text{E}-11$	volt

3.6 Cell Model Output and Equivalent Circuit Diagram for Cell Model

After the cell model parameters were obtained, the cell model response was plotted over the measured battery data. As can be seen Figure 3.6, the model output (shown in red), was in good agreement with the collected battery data. In fact, in the figure, it is hard to distinguish since the line lies directly over the measured data.

For reference, an equivalent circuit diagram representing the cell model is shown in Figure 3.7. This is similar to equivalent circuit models available provided in the literature, however, there is an addition of a dependent voltage source to model the non-linear voltage behavior of the cell near full charge and full discharge (Hartmann, 2008). In the circuit model, the cell voltage $\hat{v}(t)$ is a sum of the fixed dc voltage, V_o, the storage voltage, $V_s(t)$, the diffusion voltage, $V_d(t)$, the dependent source that captures the non-linear voltage dynamics (dependent upon the level of charge, $q_s(t)$), $V_{non-linear,region}(t)$, minus the current-dependent voltage drop across R_o.

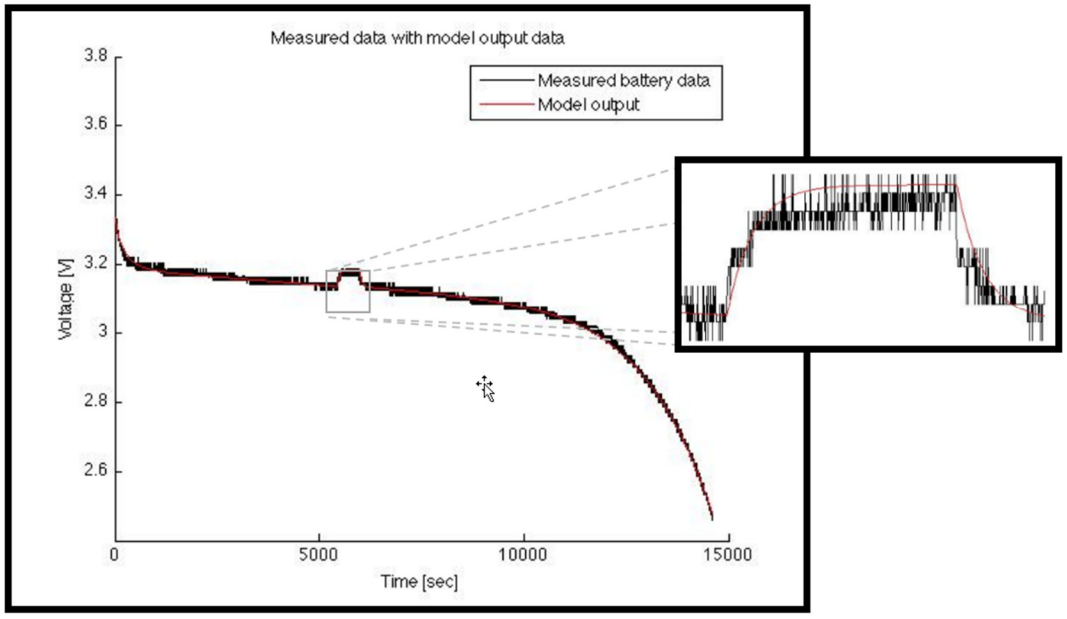

Figure 3.6: Measured battery data (black) with cell model output plotted over measured data (Red). To the right of the picture is a zoomed view of the temporary load removal.

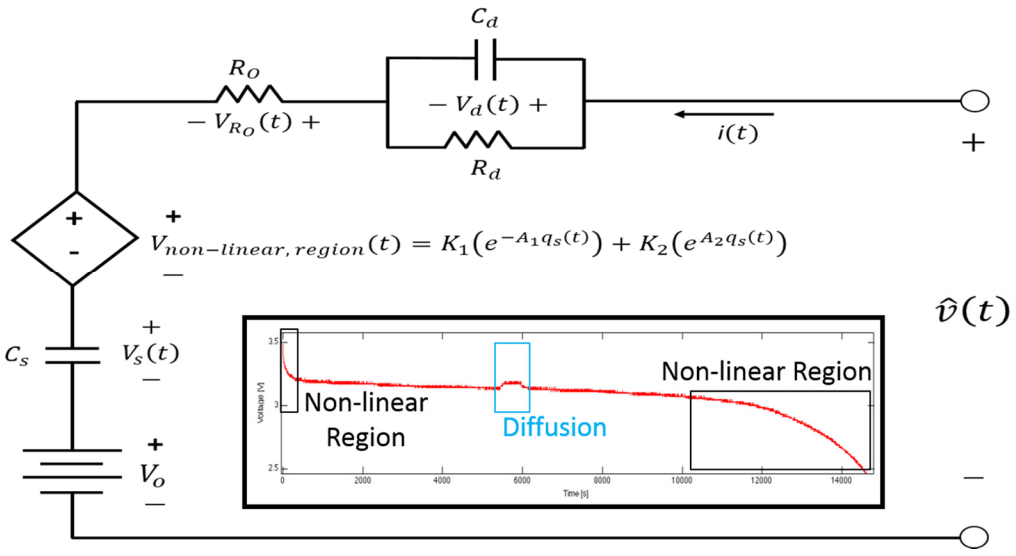

Figure 3.7: Equivalent circuit diagram for cell model with a dependent source used to represent non-linearity characteristics of cell.

3.7 Summary

In this Chapter, the background for a cell model and its various parameters were discussed. In addition, methods for implementing real-time cell models for real-time cell parameter estimates was discussed. Although such an implementation was not in the scope of this research, the benefits and abilities gained from a real-time cell model implementation were also provided. Next, the method and steps used to collect necessary cell data required to obtain parameters of the cell model were discussed. Lastly, the parameters obtained from the actual collected cell data were provided.

CHAPTER IV

CELL MODEL SIMULATIONS

This chapter will discuss the use of the cell model obtained from actual measured data. Simulations of virtual battery packs were performed and using multiple instances of the obtained cell model. First, an investigation into a hypothetical battery pack consisting of 3 cells connected in series is discussed. The purpose of the investigation is to demonstrate the effectiveness of passive bypasses in a battery pack versus a battery pack without any equalization. The end of the chapter concludes with a simulation to test the concepts of an integrated battery management and charging system. This simulation serves as a pre-cursor of which the remainder of the research was focused.

4.1 Simulations of a 3-Cell Series Connected Battery Pack

A simulation was performed on a virtual 3-cell series battery pack that consisted of several discharge and charge cycles. In this example, no bypass circuits were utilized. The goal of this simulation was to verify that without any bypass circuitry, the cells' SoC diverges over time. Since voltage depends upon the SoC, this also leads to the cells' voltages diverging over time.

The cell model parameters used to simulate the cells were kept all the same. The only difference in each cell was the initial SoC assigned to each one. Cell One was provided

a SoC 2% above Cell Two and Cell Three was provided an SoC 2% below Cell Two. It is important to note that the conditions in the simulation were chosen to demonstrate how un-balanced cell voltages within a battery pack behave. Of particular interest is whether all of the cells' voltages diverge or converge relative to one another. Also, worth mentioning is that in practice, there will be some slight variation between each individual cells' parameters (which in actual cells is due to variations in the cell manufacturing process). However, these variations will most likely only make the cells even more likely to diverge. It will be shown that just the variation in the cells' initial SoC is enough to cause divergence.

The following discussion outlines the simulation of a charging device and the simulation of a monitored discharge load. At a high-level, chargers typically provide a constant current into the battery pack being charged. The charger then monitors the overall voltage of the pack until it reaches a pre-designed constant voltage state. At this point it decreases the current to maintain a designated constant voltage. Following this, the charger either shuts off after a given period of time or waits until the charging current falls below a certain threshold. Similarly, the discharge monitoring circuitry typically monitors the voltage of the overall pack voltage until it reaches a low threshold. At this point the circuitry discontinues running the application.

Charge Cycle, Simulated Charger

- *Constant current:* The pack was charged at a constant current of 20A until the pack voltage reached 11.4V (occurring ideally when each cell is 3.8V).

- *Constant voltage:* At this point the charger maintained the constant voltage of 11.4V by reducing the current until it reached 6A.
- *End of charge cycle:* This point marks the end of the charge cycle and the discharge cycle starts shortly after.

Discharge Cycle, Simulated Monitored Load

- *Constant Discharge Current:* The battery pack is discharged at 60A until the pack voltage reaches 7.5V (occurring when each cell reaches 2.5V).
- *End of discharge cycle:* This point marks the end of the discharge cycle and the charge cycle begins again.

4.1.1 Battery Pack without Passive Bypass

Figure 4.1 shows the packs' voltage over 916 charge/discharge cycles (or approximately 4000 hours continuously running the aforementioned Charge and Discharge Cycles). A closer view of the first 10 hours of the simulation are provided in Figure 4.2. This simulation represents what is seen if only the overall battery pack voltage is monitored. Obviously, the exact voltages of the individual cells within the battery pack cannot be determined from this graph. Although the battery pack's total voltage never reached over 11.4V (3.8V x 3 cells if cells are balanced) or went below 7.5V (2.5V x 3 cells if cells are balanced), the individual cells in the pack did not stay balanced as demonstrated in Figure 4.3.

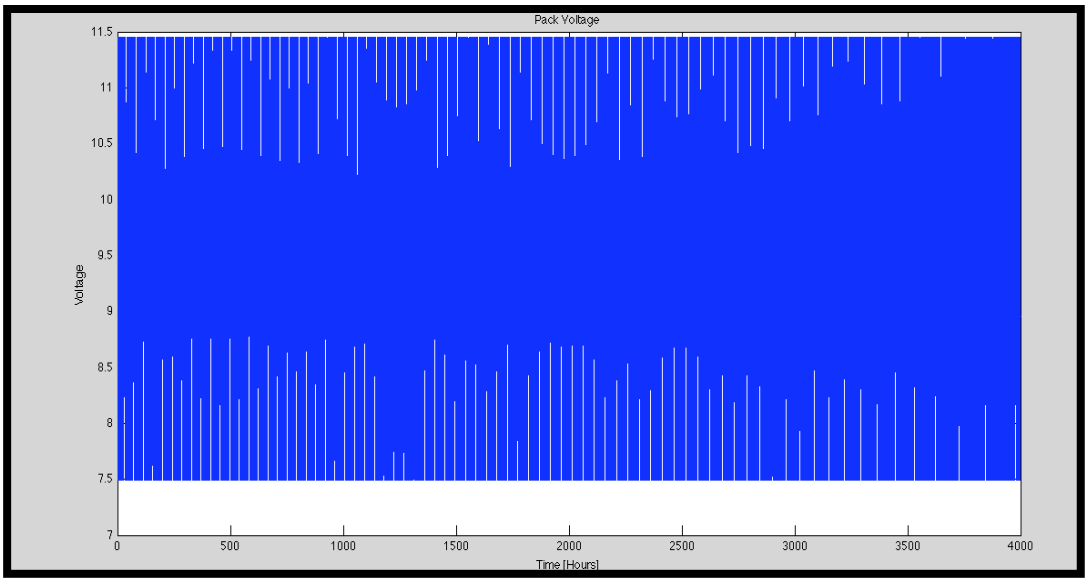

Figure 4.1: 3-cell series battery pack voltage over 916 charge/discharge cycles.

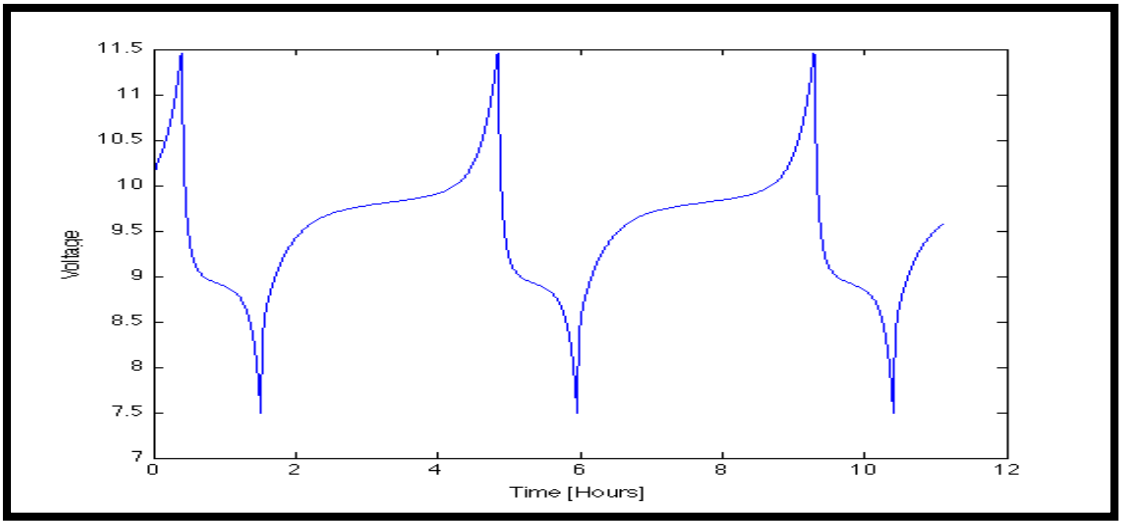

Figure 4.2: Zoomed in view of the beginning cycles of the 3-cell series battery pack simulations. The final cycles of the simulation also look similar to these.

Figure 4.3 shows the 3 individual cell voltages over the 916 charge/discharge cycles performed. The number of cycles was chosen arbitrarily in order to demonstrate the individual cell voltages of the pack diverging. Clearly, it can be seen that the cells diverge enough to allow Cell 1 and Cell 2 to reach voltages outside their manufacturer

recommended voltage operating ranges during charge and discharge, respectively. Cell 2 reaches voltages near 4.4V, while Cell 1 reaches close to 2V. To further demonstrate how the cell voltages diverged, provided are Figure 4.4 and Figure 4.5. These figures show the first few hours of the simulation and the last few hours of the simulation, respectively.

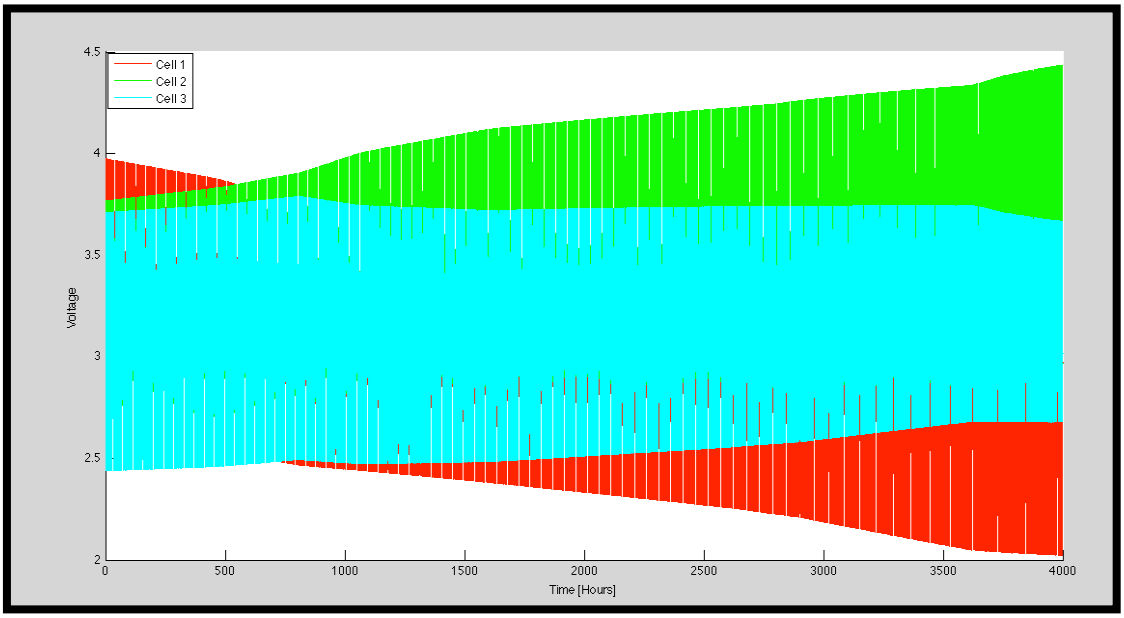

Figure 4.3: 3-cell series battery pack initially unbalanced, without balancing circuits after 916 charge/discharge cycles.

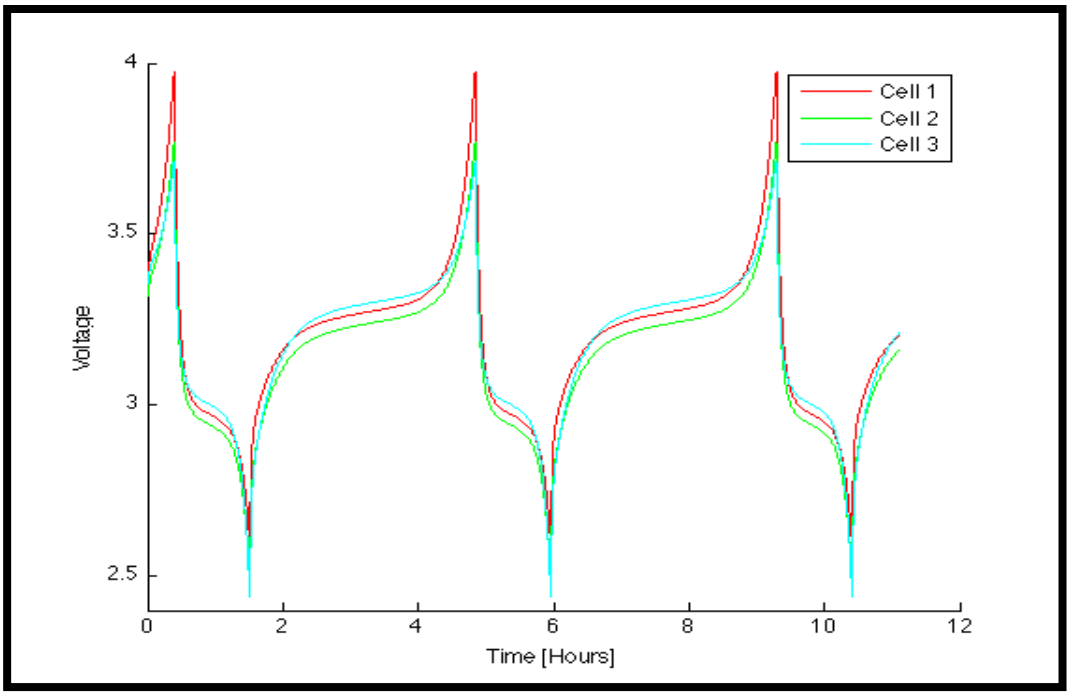

Figure 4.4: Zoomed in view of initial cycles of 3-cell simulation without bypasses.

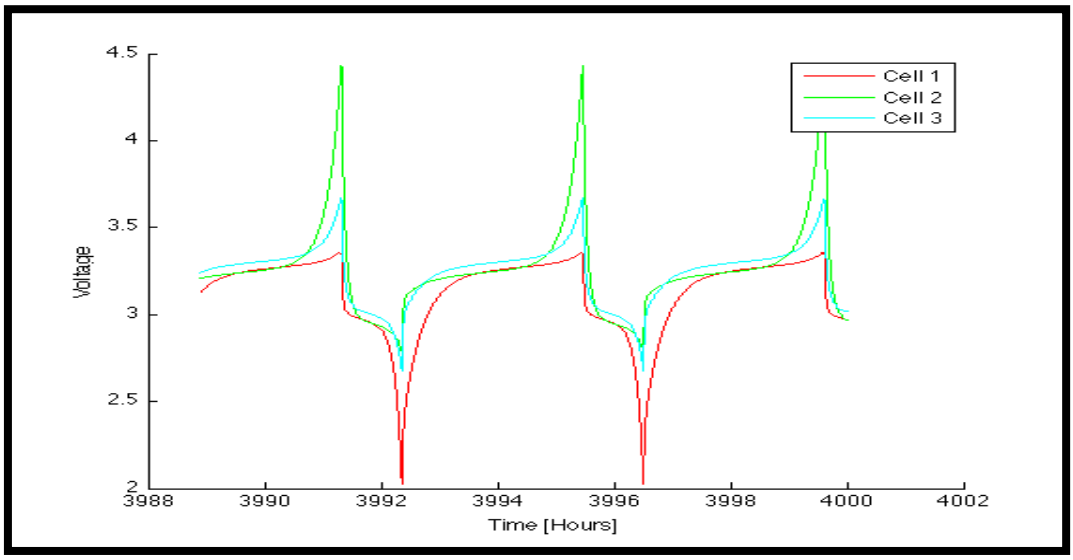

Figure 4.5: Zoomed in view of final cycles of 3-cell simulation without bypasses.

4.1.2 Battery Pack with Passive Bypass

Figure 4.6 shows the same simulation with the addition of a resistor-based current bypass on each cell. When any individual cell reached a voltage of 3.8 V, the current bypass was enabled allowing 1 amp to shunt around the bypass's corresponding cell. This provided a means to allow the higher charged cells to continue charging at a reduced charge rate and allow the lower charged cells to continue at the full rate. As can be seen, the results demonstrate the cells converged within the first 250 hours of continuously running the simulation charge and discharge cycles. Again the initial cycles of the simulation, Figure 4.7, and the final cycles of the simulation, Figure 4.8, demonstrate that the cell voltages converged. The MATLAB code for the 3-Cell pack simulations with and without bypasses can be seen in Appendix A.1.

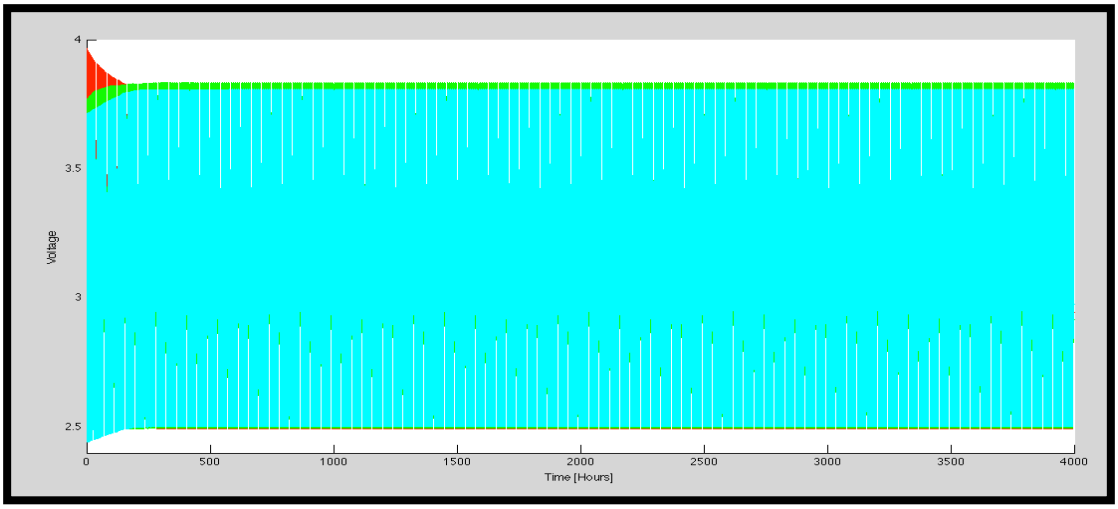

Figure 4.6: Simulations performed with the same initial conditions as the previous simulation and cell parameters but with the addition of balancing circuitry being simulated.

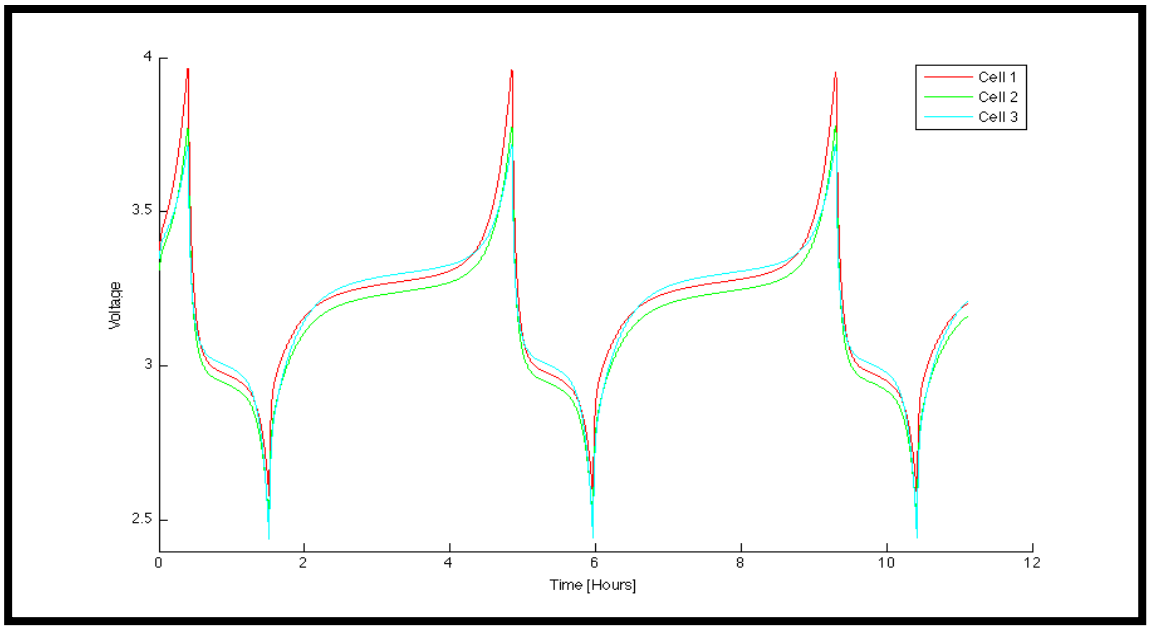

Figure 4.7: Zoomed in view of the initial cycles of the 3-cell simulation with bypasses.

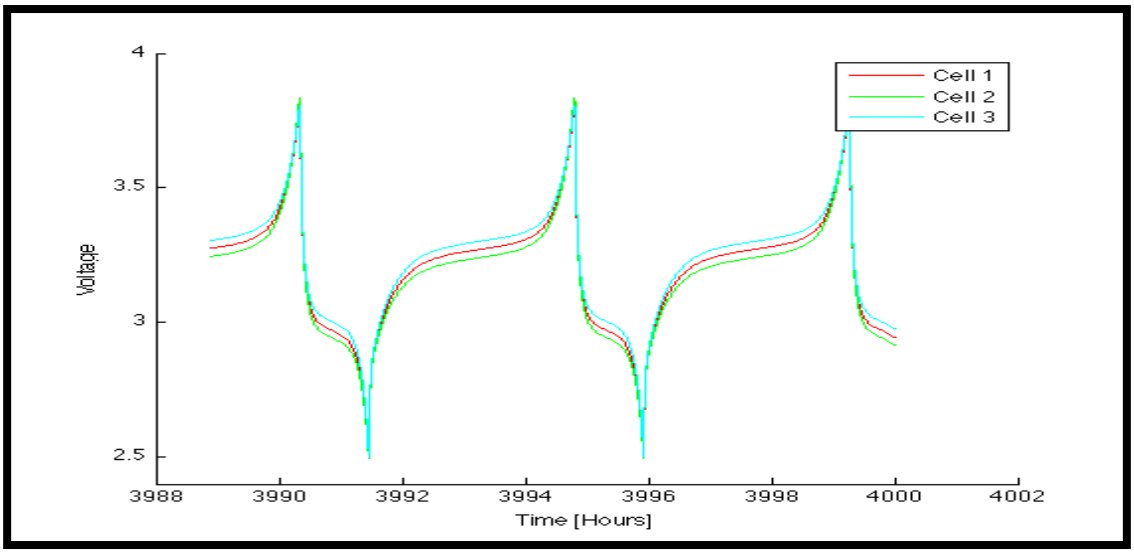

Figure 4.8: Zoomed in view of final cycles of the 3-cell simulation with bypasses.

4.2 Simulations of 10-Cell Pack

The following sections were simulations performed to determine the behavior of a 10-cell battery pack using different configurations for management. The three sets of simulations performed consisted of:

1. 10-Cell Battery Pack without any BMS and simple Charger.
2. 10-Cell Battery Pack with simple BMS and a simple Charger.
3. 10-Cell Battery Pack with BMS and integrated Charger.

The details of each of these individual simulations will be covered in the following sections. In all three of these simulations, the cells are all initialized with an idealized set of identical parameters as given in Section 3.5 (this condition is ideal in the sense that these hypothetical cells have the same exact properties which in practice may be different due to cell manufacturing tolerances). The one exception to the choice of identical parameters was the initial SoC for each cell pack arbitrarily chosen to be different. However, the values chosen for each of the cells' initial SoC are the same for each simulation set. This allows for an apples-to-apples comparison to be made on the effectiveness of the simulated management scheme. The initial SoC of each cell within the battery pack given in percentage are, 8%, 13%, 14%, 15%, 16%, 14%, 15%, 15%, 17%, and 16%. Lastly, in each of the simulations a constant-current 60A load is used during the discharge cycle.

4.2.1 Simulation with Non-Integrated Charger and without any BMS

This section discusses the simulation of the 10-Cell series-connected battery pack without a BMS and a simple charger. The simple charger charges the battery pack in constant current mode at a current of 12A. The simple charger looks at the overall voltage of the battery pack without any consideration of the individually charged cells. When the

overall voltage of the battery pack hits 42V (4.2V x 10 cells), the discharge cycle is started. The discharge cycle is complete when the voltage of the overall pack is less than 25V (2.5V x 10 Cells). This simulation reinforces the results seen in Figure 4.1 (simulation results of the overall voltage of the series-connected 3-cell battery pack) and Figure 4.3 (simulation results of the individual cell voltages of the series-connected 3-cell battery pack). The results, plotted in Figure 4.9, of this simulation will serve as a baseline for performance comparison with the other simulated methods (a closer view of a single cycle is shown Figure 4.10). Figure 4.9 shows the cell voltages and the battery pack current over 14 full charge and discharge cycles. Due to the fact that the cells are unbalanced, it is clear that specific cells are being charged such that their voltages go above the plotted threshold lines of 4.2V and 2.5V. The cell model simulations do not take into account the damage and how this alters the cell's operation, however, it is clear that each charge and discharge cycle continues to undesirably push the cell voltages outside of these bounds. The MATLAB simulation code for this simulation is provided in Appendix A.4.

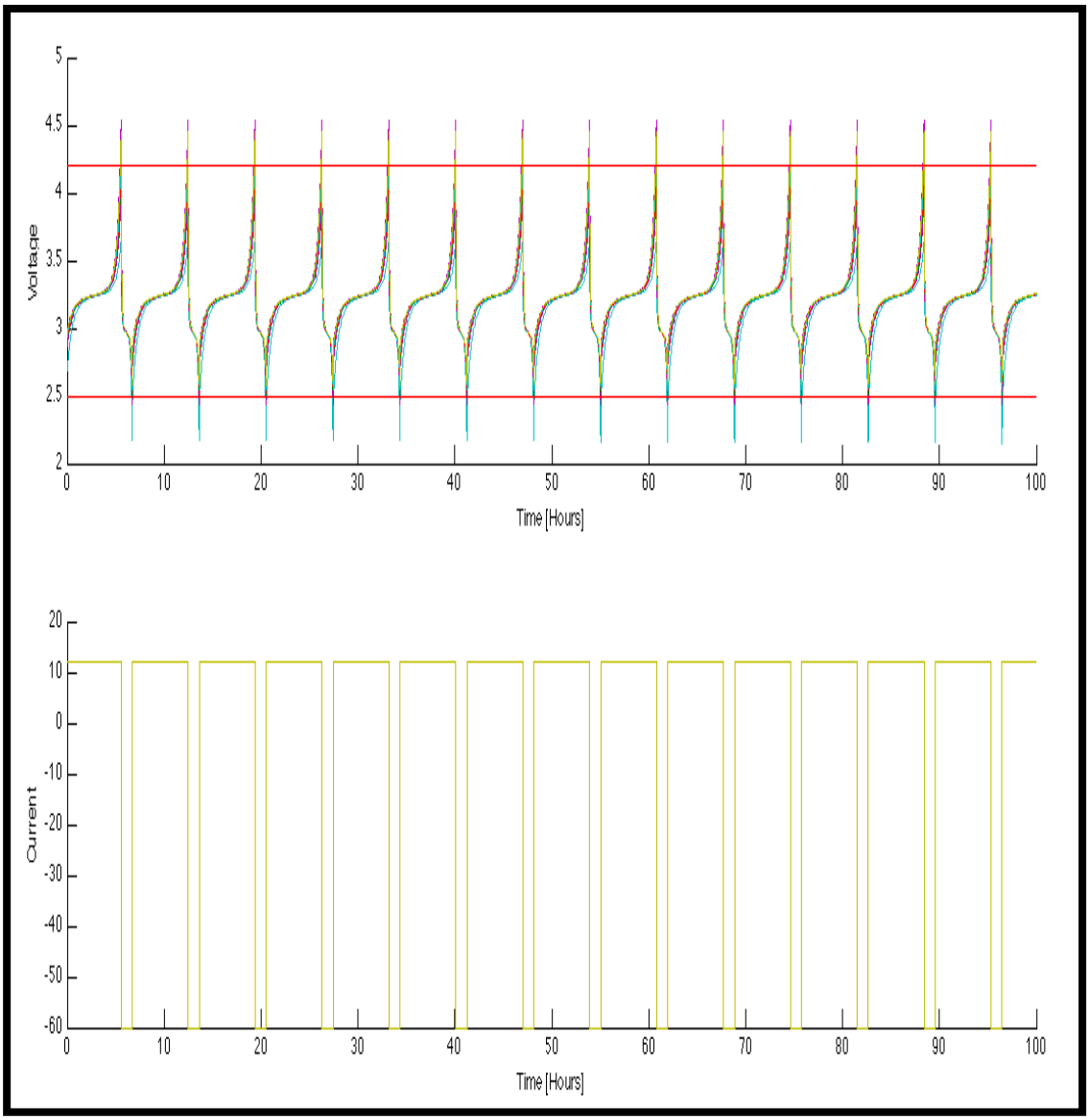

Figure 4.9: Simulations performed with the same initial conditions and cell parameters and without an integrated charger or any BMS.

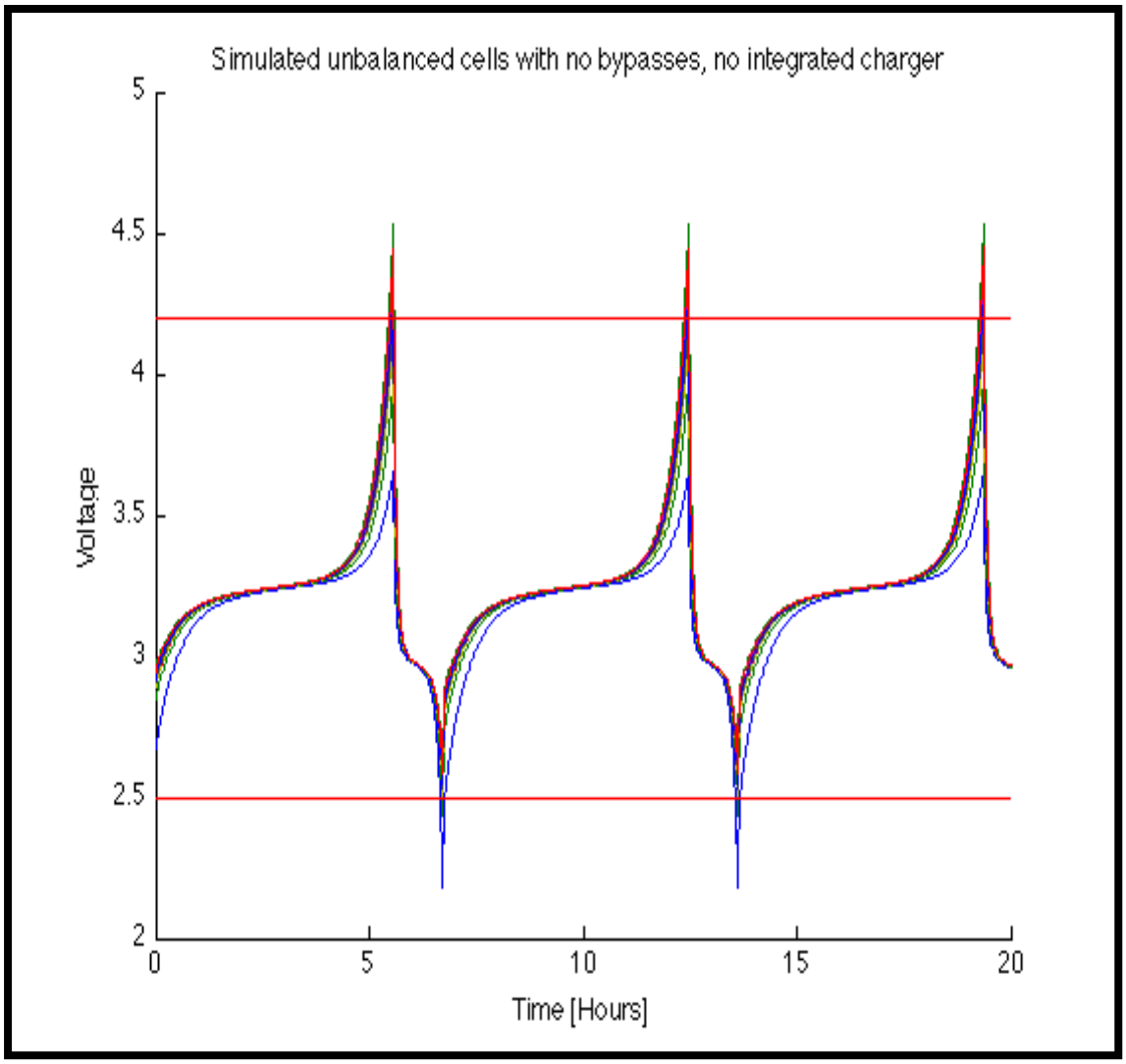

Figure 4.10: Zoomed in view of the initial cycles of 10-cell pack without bypasses.

4.2.2 Simulation with 2A Fixed Bypass and Non-Integrated Charger

This section shows results of simulations with the same 10-cell series-connected battery pack, the same simple charger, and the same load as in the previous simulation. The main difference is that constant-current 2A bypasses are simulated on each cell that turn on when the cells reach 3.8V. Figure 4.11 shows the individual cell voltages, battery pack current, and the bypass currents for each cell over 17 full charge and discharge cycles. The results are similar to the results seen in the 3-cell series-connected pack simulations. The

cells initially started off-balanced in the first few charge/discharge cycles, going outside the 4.2V and 2.5V threshold lines. However, as the bypasses are able to provide more balancing during each charge cycle, the cell voltages begin to converge. Towards the end of the simulation as the cells become more balanced, the cell voltages are able to stay within the threshold lines. A comparison between the unbalanced cell voltages at the beginning of the simulation and the balanced cell voltages at the end of the simulation can be seen in Figure 4.12 and Figure 4.13, respectively. The MATLAB simulation code for this simulation is provided in Appendix A.2.

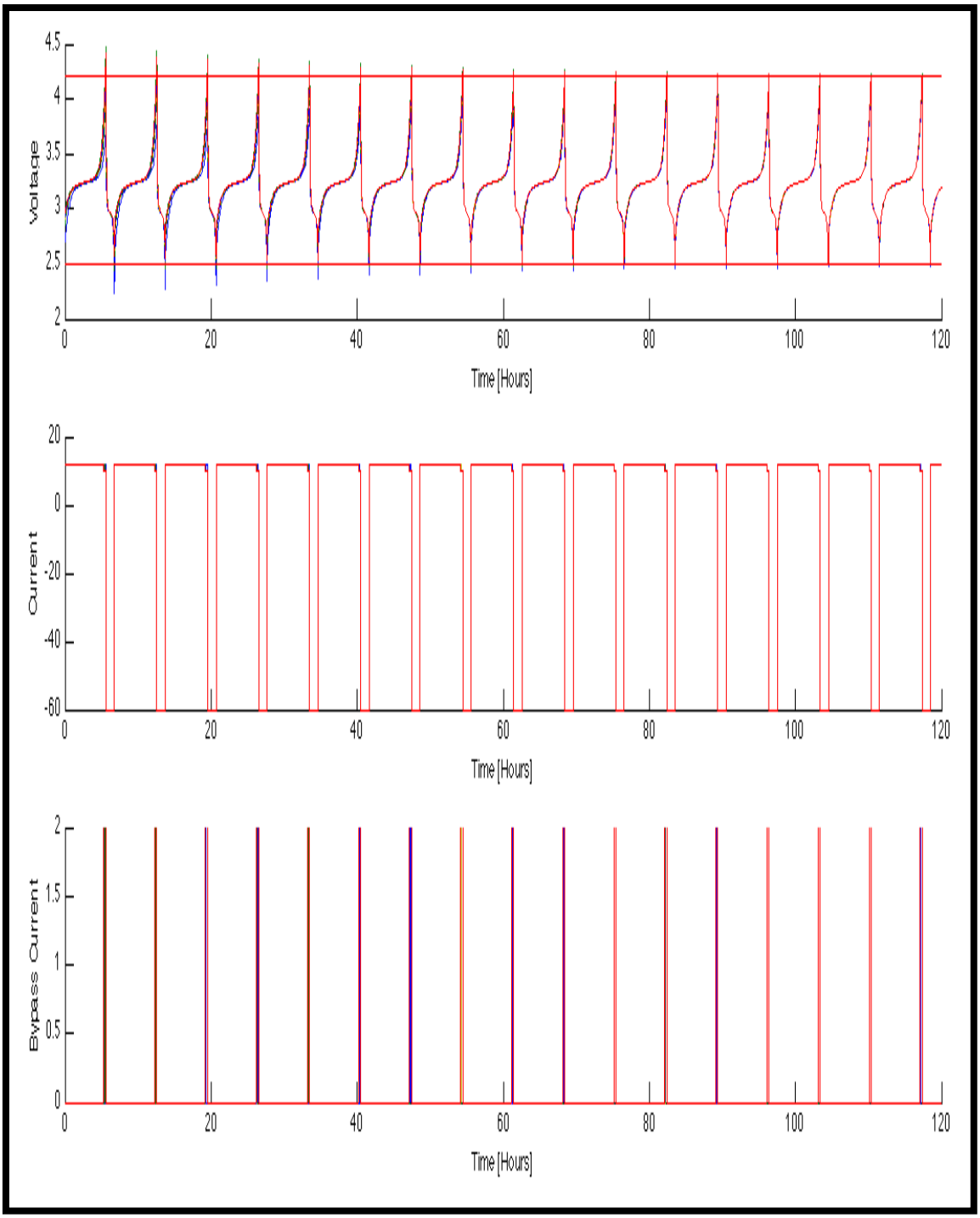

Figure 4.11: Simulations performed with the same initial conditions and cell parameters but with the passive bypass and non-integrated charger.

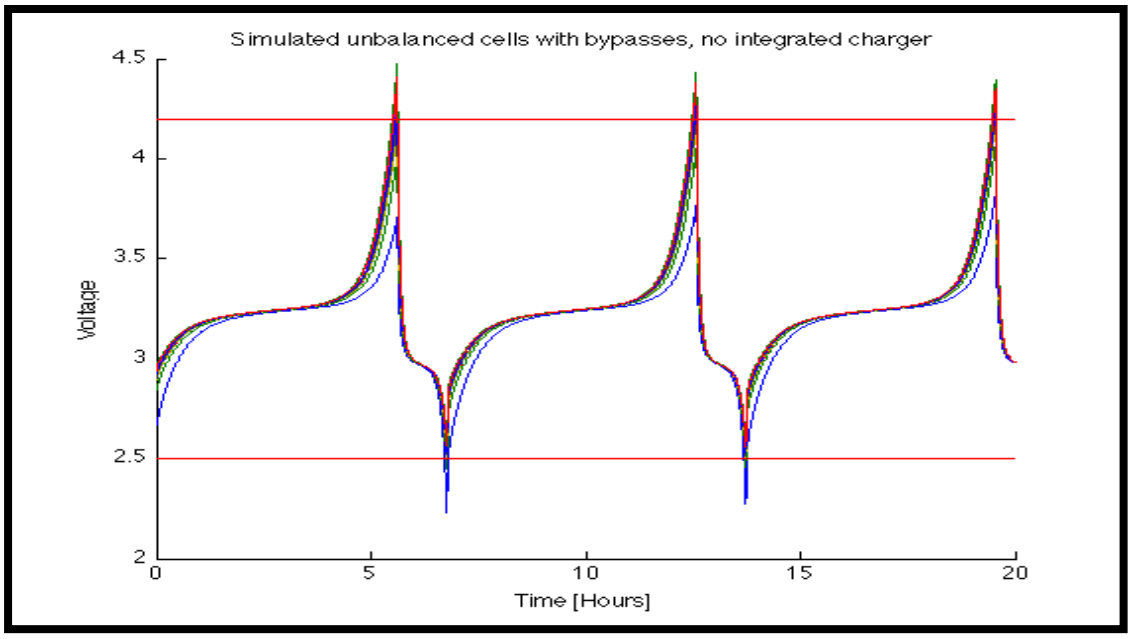

Figure 4.12: Zoomed in view of the initial cycles of the passive bypasses-only-simulations.

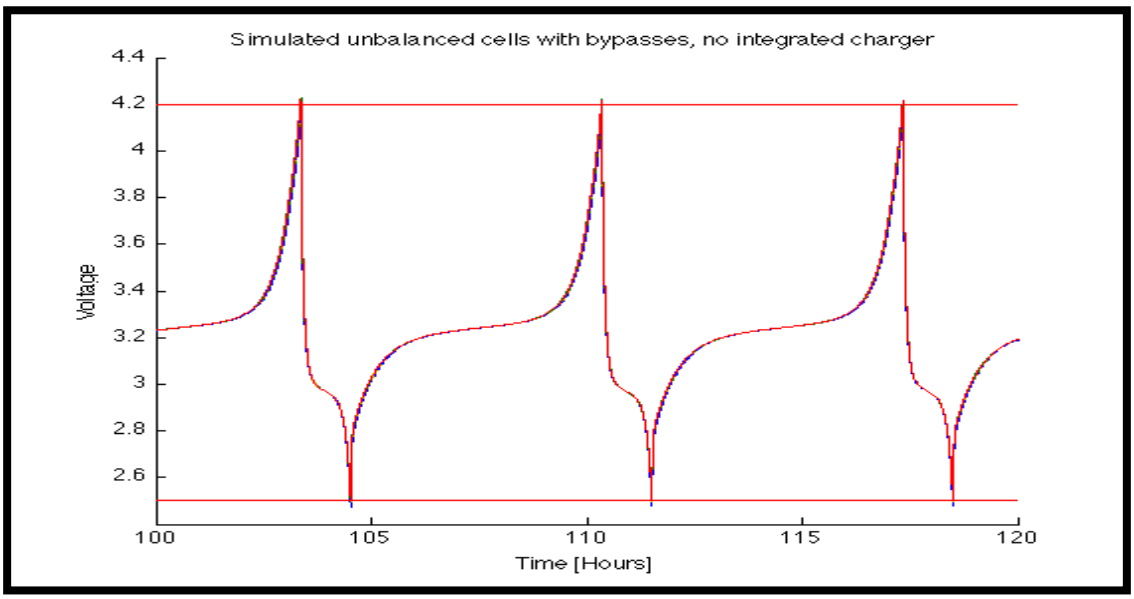

Figure 4.13: Zoomed in view of the final cycles of the passive bypasses-only-simulations.

4.2.3 Simulation with BMS and Integrated Charger

This section discusses the simulation of the proposed 10-Cell series-connected battery pack with BMS and integrated charging system as discussed in Section 2.7. The integrated charging system is simulated to charge at 12A. This constant current mode continues until any cell reaches 3.8V. Upon this event, the simulation will begin to shunt current around that cell to try to maintain the 3.8V. If the shunting current reaches 2A for any given cell, the max shunt current value, the simulation reduces the charging current by 2A. This effectively reduces the charging current for the cell that originally reached 3.8V to a charging current that is 2A less, with an additional possible reduction of up to 2A shunted via the simulated bypass circuitry. Shortly after decreasing the charging current by 2A and with the additional shunt current of 2A, this caused the cell's voltage to decrease. The simulated shunt current will be controlled and decreased to try to maintain 3.8V. After enough charging time has elapsed at the new lower charging current, the cell voltage will begin to increase again. This in turn will cause the shunt current to increase again in attempt to maintain the 3.8V. Once again upon the cell reaching its maximum shunt current of 2A, the whole cycle repeats beginning with the BMS communicating to the charger to reduce its current by 2A. This will continue until the charging current is reduced to 1.8A. At this value, the maximum shunt current is larger than the charging current, and the BMS can easily maintain each cell at the voltage of 3.8V via the bypasses.

The goal of the aforementioned charging method is to allow the charging current to remain as large as possible throughout the charging process. This allows the lowest charged cells to continue charging at the highest possible rate, and only reducing it as necessary, to protect the highest charged cell.

The results in Figure 4.14 show the individual cell voltages, battery pack current, and the bypass currents for each cell over 7 full charge and discharge cycles. A closer view of the first cycle and the balancing of Figure 4.14 is shown in Figure 4.15. The cell voltages are able to stay within the thresholds of 4.2V and 2.5V much better than the previous simulated methods. This is due to the BMS's ability to control the charger such that the BMS can continue its cell balancing operations as desired. The MATLAB simulation code for this simulation is provided in Appendix A.1.

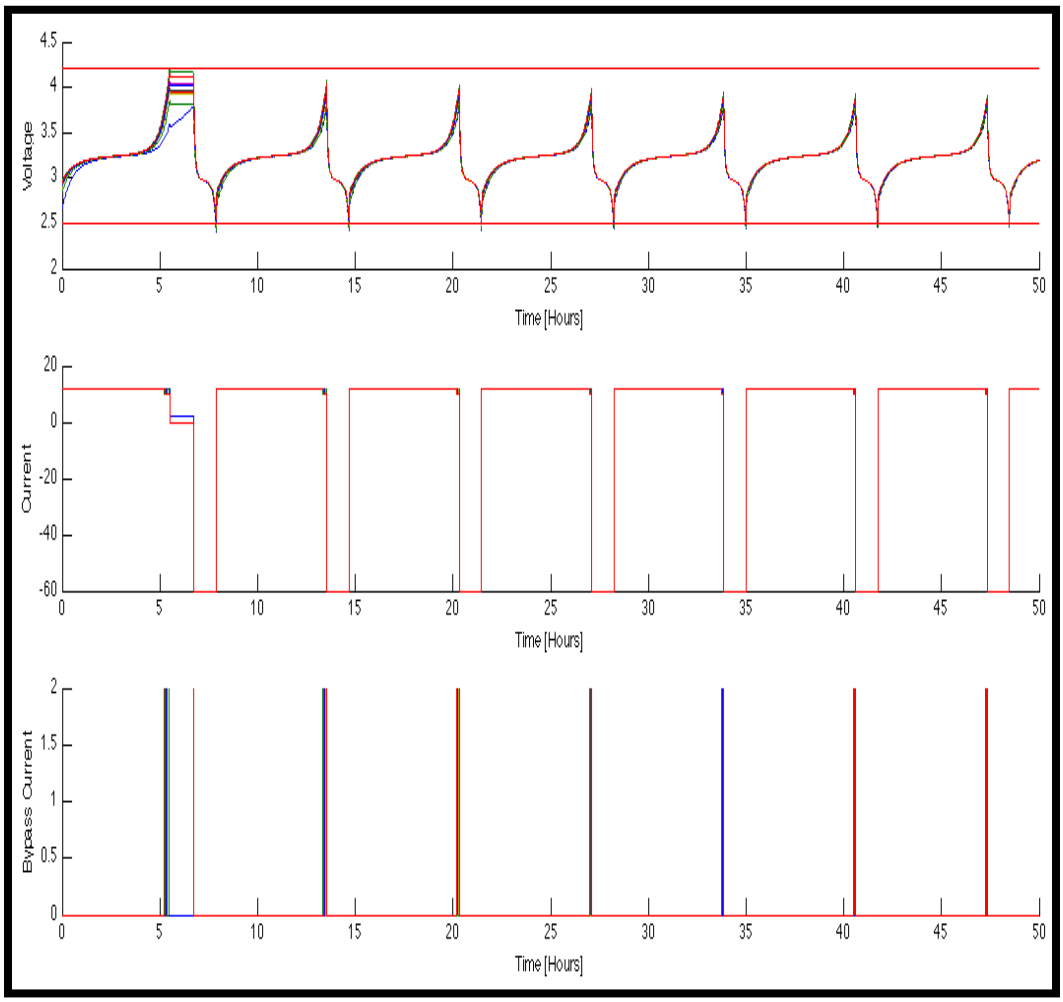

Figure 4.14: Simulations performed with the same initial conditions and cell parameters but with the integrated charger and with BMS.

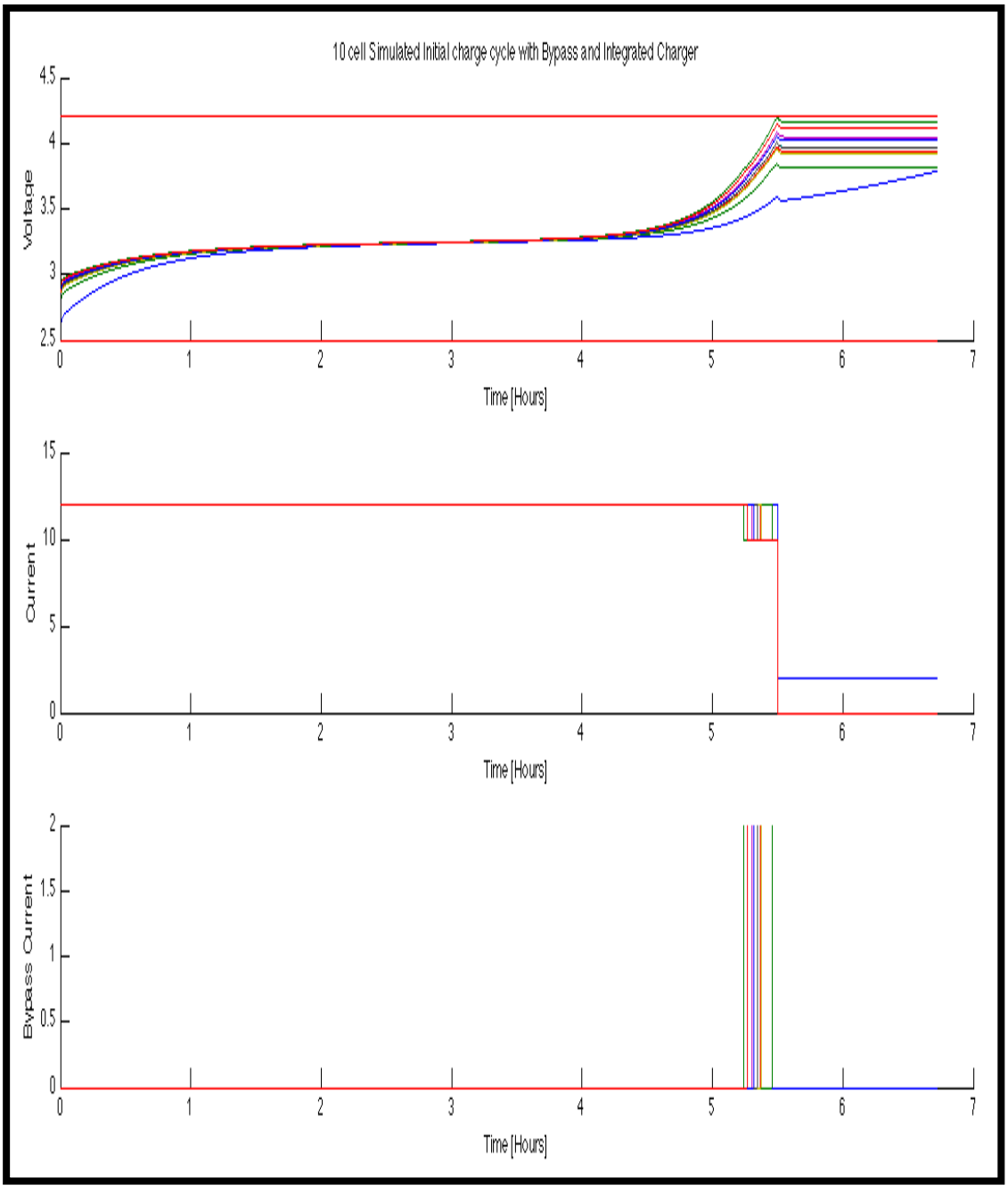

Figure 4.15: Zoomed in view of the first cycle of integrated charger and BMS simulation.

4.3 Summary

This section discussed battery pack simulations performed with the cell model obtained in CHAPTER III. The simulations were done to compare a hypothetical 3-cell battery pack with and without passive bypasses. The battery packs were cycled multiple times in the simulation to demonstrate that without passive bypasses the cells drift further and further out of balance from one another. The pack with the passive bypasses initially went slightly out of the desired voltage range, but then converged over time.

Also simulated in this section were 3 different management schemes for a 10-cell series-connected battery pack. The 3 different schemes simulated consisted of: a system without any management and a simple charger, a system with a fixed 2A bypass and simple charger, and lastly, a system with an integrated charger and BMS. These simulations were conducted to show the benefits gained from integrating the charger and BMS over other simulated methods. The main benefit is that the BMS may use the integrated charger interface to support cell balancing. The simulations show that by analyzing each cells' voltage and reducing the overall charging current as needed, the BMS can better maintain cell operating voltages and better equalize the SoC of each cell within the battery pack.

CHAPTER V

HARDWARE FOR BMS IMPLEMENTATION

This chapter will discuss the hardware designed and implemented for the integrated charger and BMS. The battery pack, which the proposed battery management system was implemented on, consisted of 10 series-connected $LiFePO_4$, 60Ah, 3.3V (nominal) cells. One of these cells is shown in Figure 5.1. The resulting battery pack formed from these cells creates a 33.3V, 60Ah pack. This is roughly 2kWh, which for reference is enough energy to run a 60W bulb continuously for 33.3 hours.

Figure 5.1: Single 60Ah $LiFePO_4$ Cell.

5.1 Overview of Hardware for Integrated Charger and BMS

The proposed BMS for this battery pack system consisted of 10 PCBs: 9 ICMU and 1 ICMU/BPMU hybrid. The ICMU's main function is to monitor and provide equalization capability for one specific cell. In addition to these tasks, each ICMU is also responsible for communicating information to the BPMU. This information includes cell voltage, cell temperature, and any fault detections such as overvoltage or undervoltage conditions. The BPMU simply compiles all this information, and based upon the individual cells' conditions, makes decisions about how the charger and load controller interact with the battery pack.

Each of these printed circuit boards (PCBs) were mounted directly onto one of the 10 cells in the battery pack. The placement of each PCB onto a given cell was arbitrary with the exception of the ICMU/BPMU hybrid PCB. The ICMU/BPMU hybrid PCB was placed on the cell at the lowest voltage potential with respect to the negative terminal of the entire battery pack. The reasoning behind this was to allow the BPMU to be at the same ground potential of the charger when the battery pack was connected to the charger device. This allowed the BPMU to communicate with the charging device without the requirement of electrical isolation in between the systems. Although it was not necessary for functionality, isolation between the BMS and charger is ideal. The benefits of isolation include mitigation of electrical noise being transmitted between the two circuits, as well as provision for a level of fault protection for the logic circuitry.

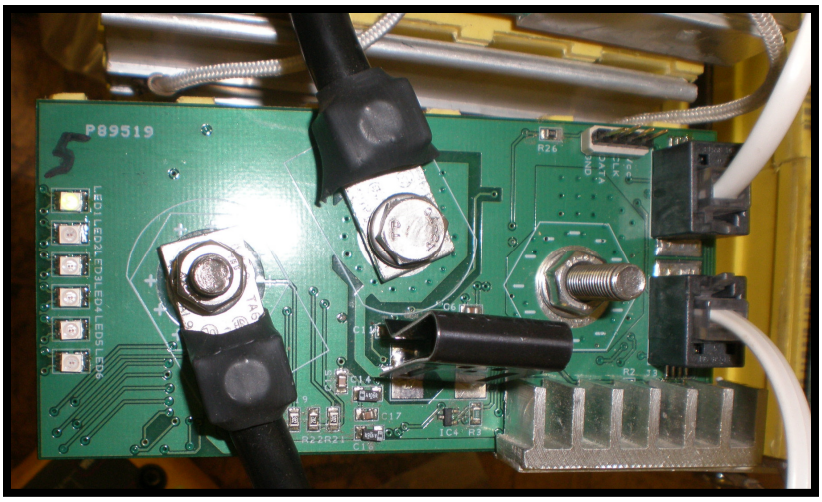

Figure 5.2: Individual Cell Management Unit pictured mounted on a single $LiFePO_4$ Cell.

5.1.1 Hardware for Individual Cell Management Unit (ICMU)

In this BMS implementation, the ICMUs measure the cells' voltage, current, and temperature. Powering the ICMUs was accomplished by designing PCBs that mounted directly onto each of the individual cells' terminals. Each ICMU is responsible for monitoring its own cell and detecting if an undesirable condition or use of the cell will cause damage to it. Such conditions as discussed earlier are overvoltage, undervoltage, and overtemperature.

The ICMU periodically transmits its own cell's information (estimated SoC, voltage, current, and temperature) to the BPMU via an isolated serial communication bus. In addition to this information, the ICMU also transmits information regarding to if any overvoltage, undervoltage, or overtemperature conditions exists. The BPMU then compiles all information from all the individual ICMUs and monitors for any cell damaging conditions. If a condition is detected, the BPMU will take action to either stop the charging or discharging process or provide fault indication to a user.

In the event of an overvoltage condition during charging of the battery pack, the ICMU also has the ability to bypass up to 2A of current around its respective cell. In doing so, this allowed the ICMU to slow down the charge rate of the cell it was managing. This allowed the other cells in the series-connected pack to continue charging at a faster charging rate and therefore provide a means of balancing the cells.

5.1.2 PCB Connection

PCBs were mounted on top of the $LiFePO_4$ cells via the positive and negative threaded terminals. The PCB's Top copper pour is shown in Figure 5.3 and the Bottom copper is shown in Figure 5.4. Further details and schematics of this PCB's implementation will be further covered in the following sections. However, the connections of the PCBs to the cells will be discussed here first. Proper mechanical connections to the cells were vital for safe usage of the cells. If the connections are not made such that they are tight and secure, they may become loose with mechanical vibrations overtime. This in turn causes the connections themselves to become a high impedance during the cell's usage. This will result in high power loss to occur at these loose connections in the form of heat. The heat generated can be enough to melt both the cell's outer enclosure and also damage the PCB.

In order to allow the boards to be mounted, M6 threaded rods are screwed into each of the cell's terminals. To allow the bolt to be tightened further a nut was placed on the threaded shaft and tightened with a crescent wrench until it was flat against the cell's outer terminal Figure 5.5. Figure 5.6 shows the same PCB from the top view.

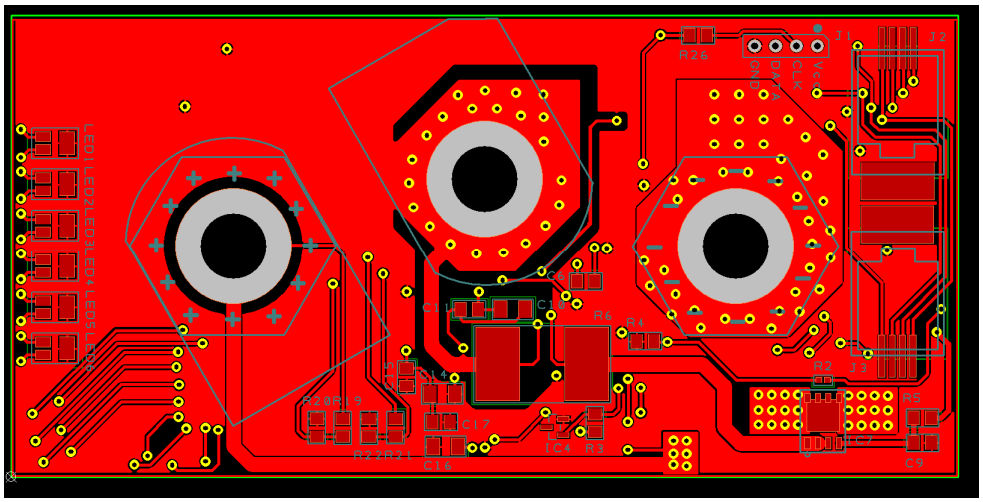

Figure 5.3: Top side of ICMU PCB.

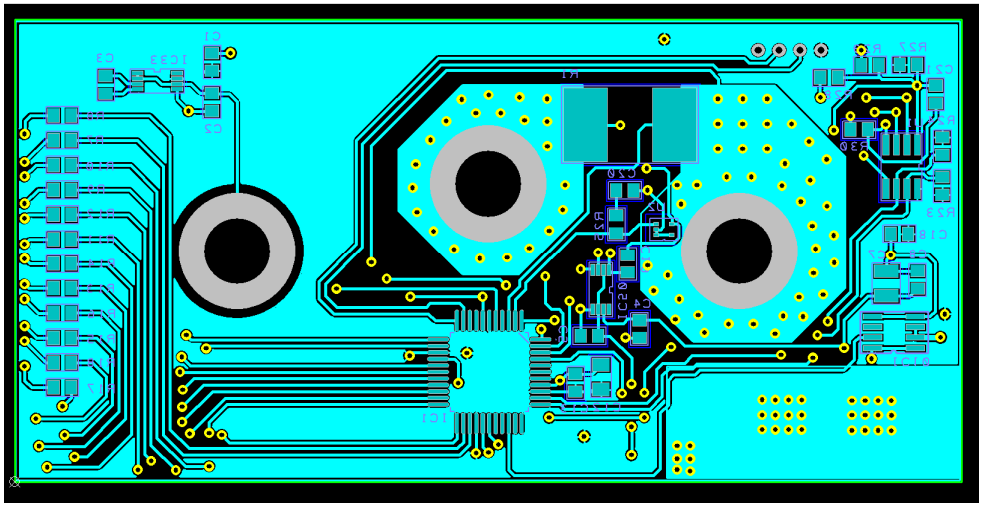

Figure 5.4: Bottom side of ICMU PCB.

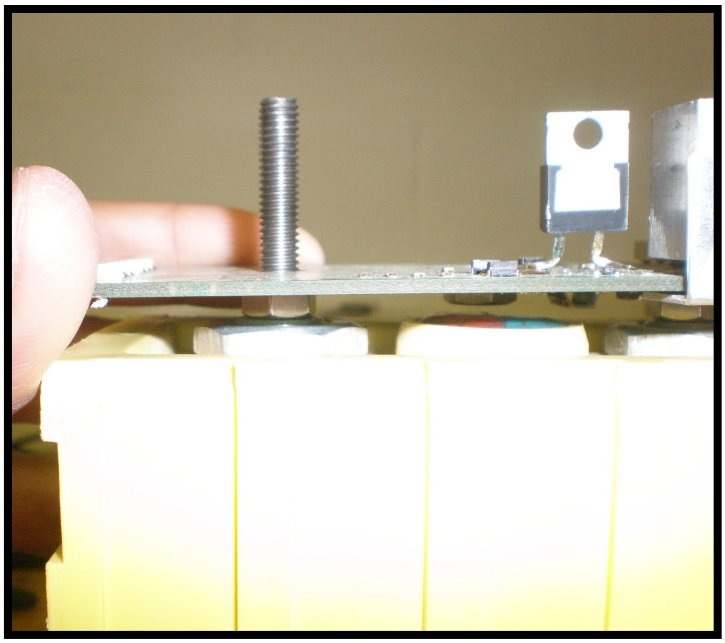

Figure 5.5: Cell with M6 threaded bolt inserted with nut.

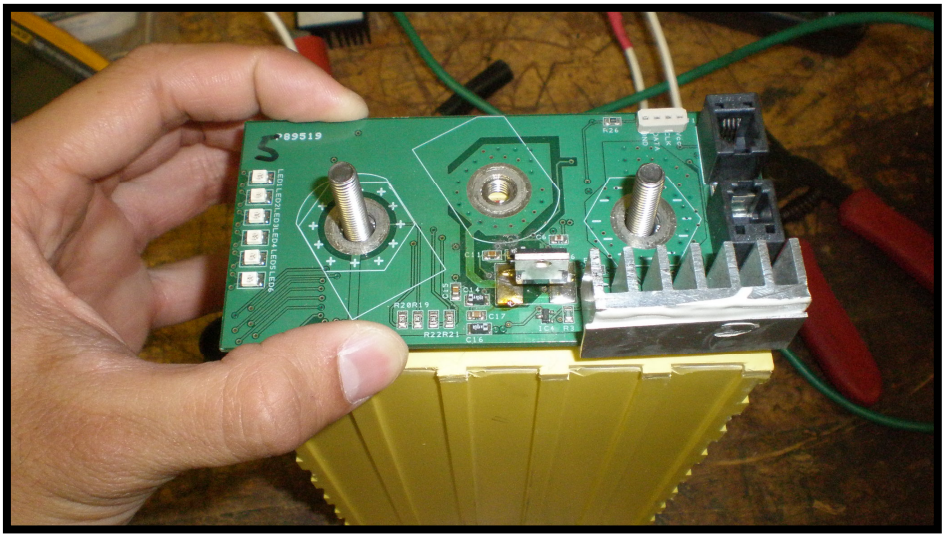

Figure 5.6: Top View of ICMU board on a single $LiFePO_4$ Cell before being tightened down with nuts on the threaded shafts.

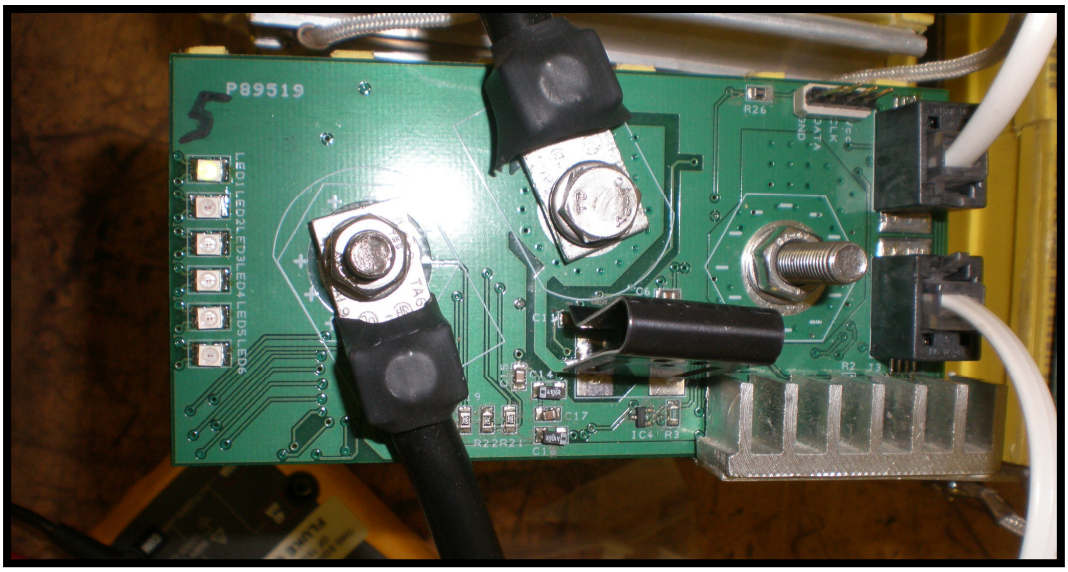

Figure 5.7: Circuit board tightened down onto battery terminals and also shown are connections to adjacent cells in the series connected cells.

5.1.3 ICMU Power Circuitry

The ICMU circuitry is powered with 3.3 volt and 5 volt power rails. These are provided via two MCP1253 charge pump integrated circuit chips. Both of these chips are directly powered off the cell the ICMU is connected to. As can be seen with the schematics pictured in Figure 5.8 and Figure 5.9, either a 3.3 volt or 5 volt output may be generated from the chip by either pulling the SELECT pin on the chip high or low, respectively. The 3.3 volt source is used to power the ICMU microcontrollers, two temperature sensors, and an isolated I2C chip. The 5 volt source is used to power a MOSFET gate driver integrated chip. These devices will be covered further in the sections to follow.

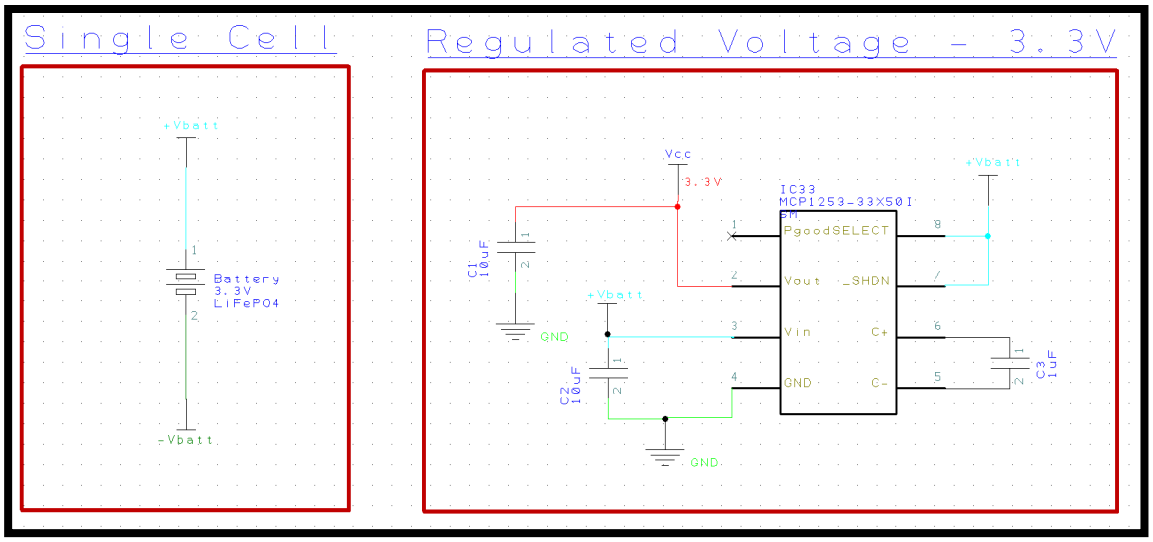

Figure 5.8: Schematic: Single $LiFePO_4$ Cell and 3.3V bus derived from charge pump.

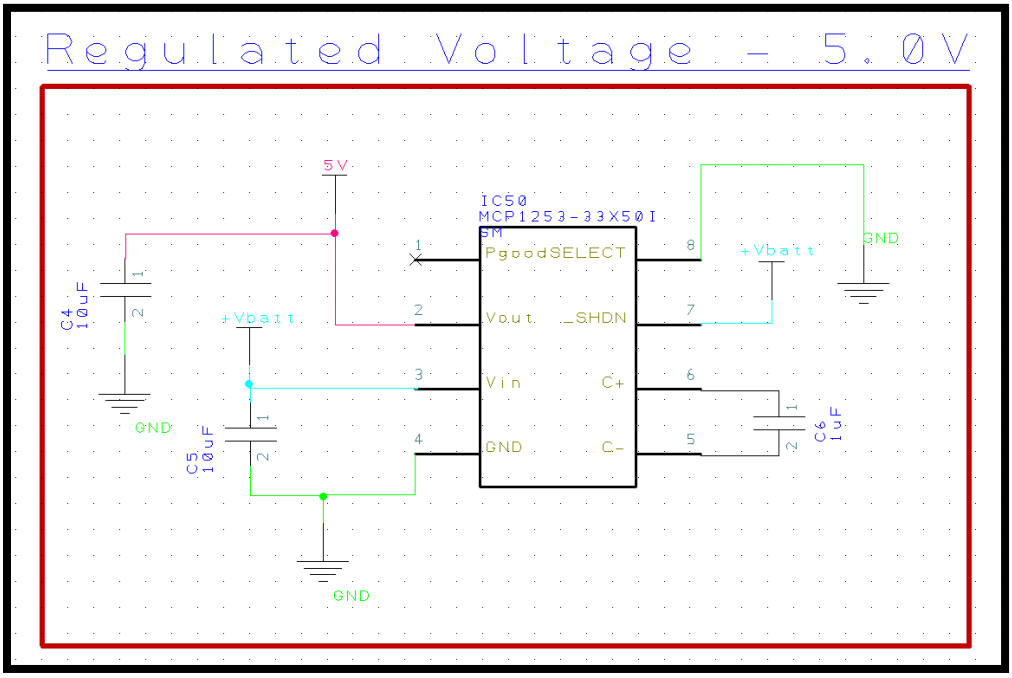

Figure 5.9: Schematic: 5 volt bus derived from charge pump.

5.1.4 ICMU Microcontroller

The Atmel ATXMEGA16A4 is a 16-bit processor that was selected primarily for its ADC features and low power capabilities (Figure 5.10). The ADC is capable of differential analog measurements. Once the differential measurements are brought into the device, the output from the differential measurement can then be applied to an internal programmable gain amplifier. The microcontroller's ADC device was implemented with an external 2.048 volt shunt reference. Utilizing a 2.048 volt reference and placing the ADC in "signed differential mode" with 12-bit resolution conveniently results in analog conversions that correlate with 1mV per 1-ADC unit. In other words, an ADC reading of 2047 (decimal value) from the ADC unit corresponds with a voltage of 2.047V. An ADC reading of -2048 in turn correlates with a -2.048V differential measurement. The interface to the microcontroller consists of I2C communication, multiple analog readings, in-circuit programming port, and a digital PWM output. These interfaces will be individually covered in the sections to follow.

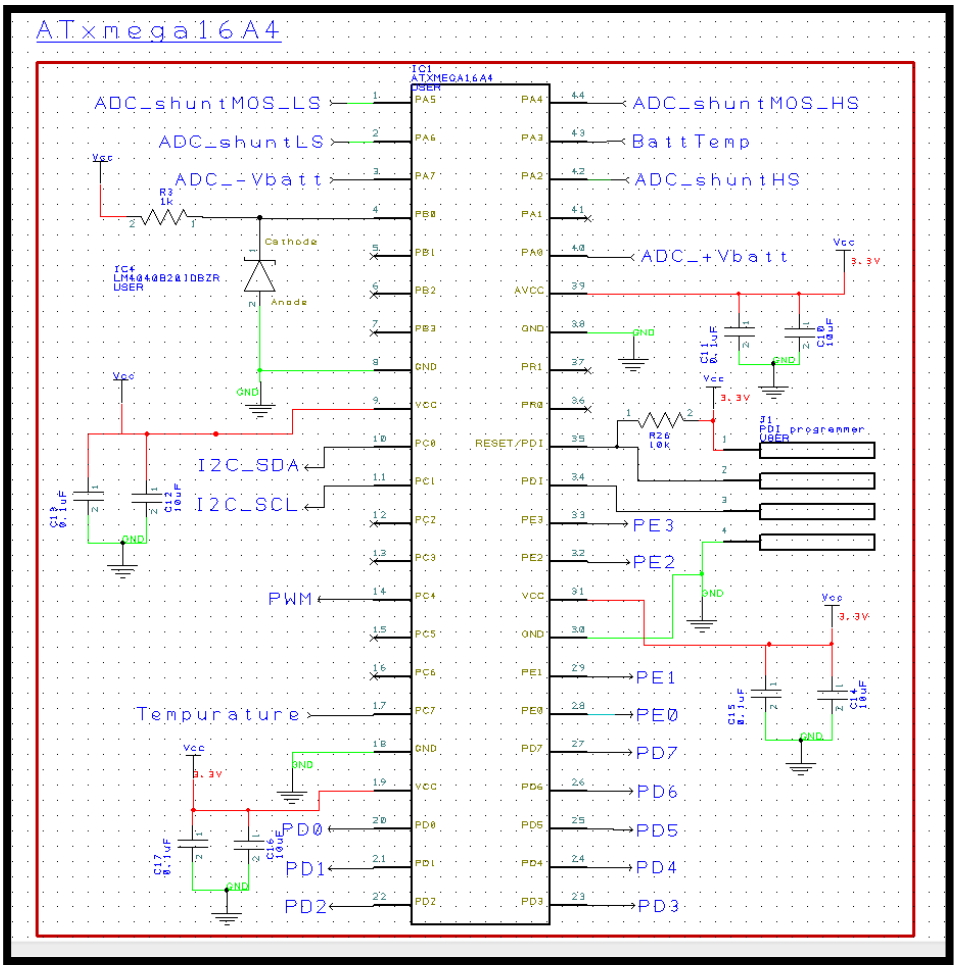

Figure 5.10: Schematic: Microcontroller schematic for ICMU PCB.

5.1.5 Implementation of Voltage and Current Monitoring

The voltage monitoring is implemented with a pair of relatively high impedance voltage dividers as show in Figure 5.11. Two identical dividers are used to ensure that the differential measurement is scaled to each input of the differential amplifier. The inputs to these voltage dividers are the positive and negative terminals of the cell that the ICMU is connected to. The resistor values of the divider was chosen to scale the cells' maximum expected voltage down to the maximum voltage of the microprocessor's ADC (the value of the external voltage reference of 2.048V). In addition, a 0.1uF ceramic capacitor is

placed across the stepped down voltage to serve as a low pass filter. An undesirable effect of this implementation method is that it requires a constant current to be drawn from the cell at all times. Since the cells have a relatively large capacity compared to the current that is drawn from the divider, it can be considered negligible. This is under the assumption that battery pack will be charged on a regular basis and not stored for a long period of time. Storage for a long period of time with any additional current draw will certainly bring the SoC of the battery pack down faster.

Also pictured in Figure 5.11, is a resistive shunt that was utilized to measure the current in and out of the cell. As can be seen in Figure 5.7, the PCB has a third terminal in the center. This third terminal has a shunt connected to it, which in turn is connected to the cell's negative terminal. This extra terminal and the placement of the shunt allows the current to be measured going in and out of the cell's negative terminal. In the schematic of Figure 5.11 the value of 2 milli-ohms is shown as the original designed value, but after testing, a 6 milli-ohm shunt was used to scale the maximum current measuring range to a more preferable value. This shunt was utilized to create a voltage drop that correlates with the current going in and out of the cell. To get the voltage measurement on the shunt, a differential analog measurement was taken across the shunt. This differential measurement was then sent through the programmable gain amplifier with a gain of 8. Utilizing the programmed gain and the chosen 6 milli-ohm shunt, provided a maximum theoretical reading range of -42 to 42 amps correlating with the range of voltages coming out of the programmable gain amplifier of -2.048 to 2.048V. Figure 5.12 shows the actual implementation of the PCB.

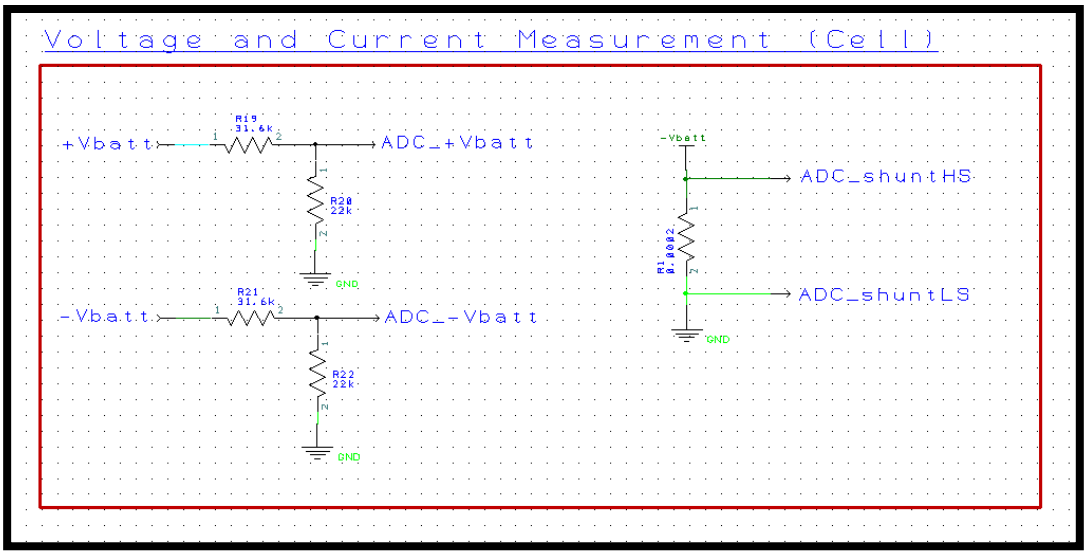

Figure 5.11: Schematic: Voltage divider to scale cell voltage for microcontroller's ADC on ICMU PCB and measure the current going in and out of the managed cell.

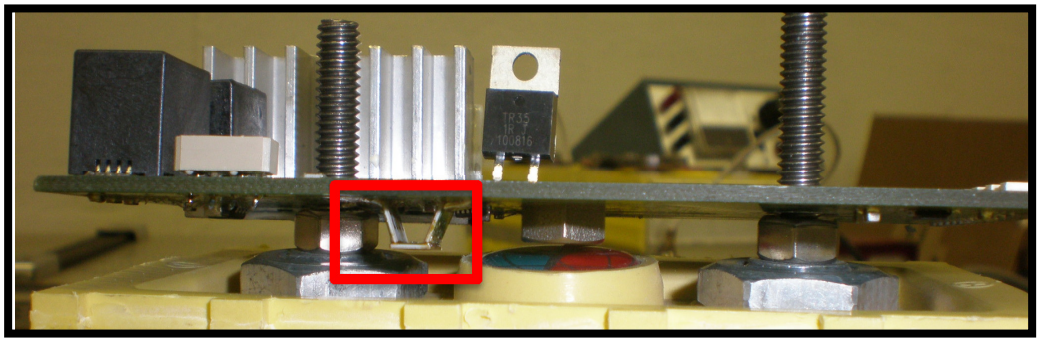

Figure 5.12: Current Shunt used to measure system current is pictured in red box.

5.1.6 Implementation of Current Bypass Circuit

The main components of the bypass circuitry consisted of a MOSFET switch in series with a 1 ohm, 25 watt power resistor. This circuitry, shown in Figure 5.13, was used when an overvoltage condition was detected during a charge cycle.

The current bypass circuitry was designed to allow current to be bypassed around a cell in the range of 1-to-2 amps. This was implemented by operating the MOSFET in its

linear region. In other words, the MOSFET was used as a voltage controlled resistor. In order to measure how much of the current was bypassed, the differential voltage drop across the 1-ohm power resistor was measured with the ICMUs microcontroller's ADC.

The voltage control on the gate of the MOSFET was implemented using the output from a 20.8 kHz PWM signal fed into a low-pass RC filter. The PWM was generated from the ICMU's microcontroller PWM peripheral. The RC filter consisted of a 10kohm resistor and 0.1uF ceramic capacitor as seen at the MOSFETs gate in Figure 5.13. These two provide a cut-off frequency of 159 Hz with the transfer function

$$G(s) = 1000/(s+1000)$$

To control the amount of current to be bypassed, a proportional controller was implemented. The controller's reference input was the desired maximum voltage the cell being protected was allowed to reach. If the voltage of the cell increased beyond the maximum desired voltage the ICMU's microcontroller increases the PWM duty cycle. This increases the voltage on the capacitor in the low-pass filter, which in turn decreases the MOSFET's drain to source resistance.

The heat generated from using the MOSFET as a voltage-controlled resistive bypass, was dissipated via a heat-sink. To ensure that the heat generated did not rise to an excessive level, a temperature sensor was used. The temperature sensor was simply a thermistor placed near the bypass MOSFET that was placed within a voltage divider with a fixed resistance. The actual implementation of this is shown in Figure 5.14.

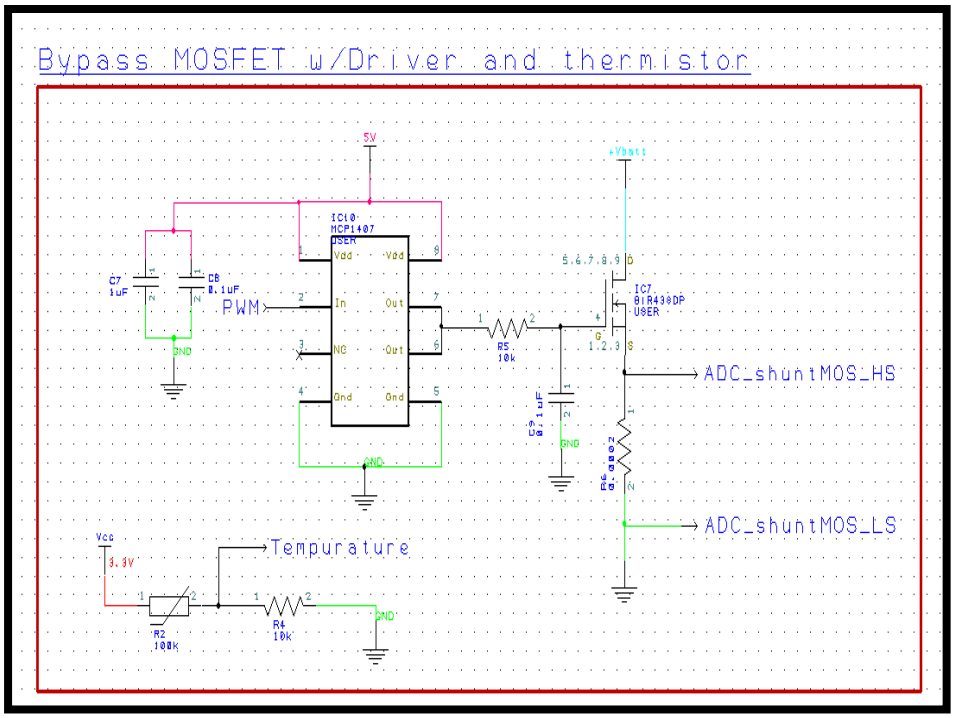

Figure 5.13: Schematic: Bypass MOSFET circuit and bypass's overtemperature Sensor on ICMU PCB.

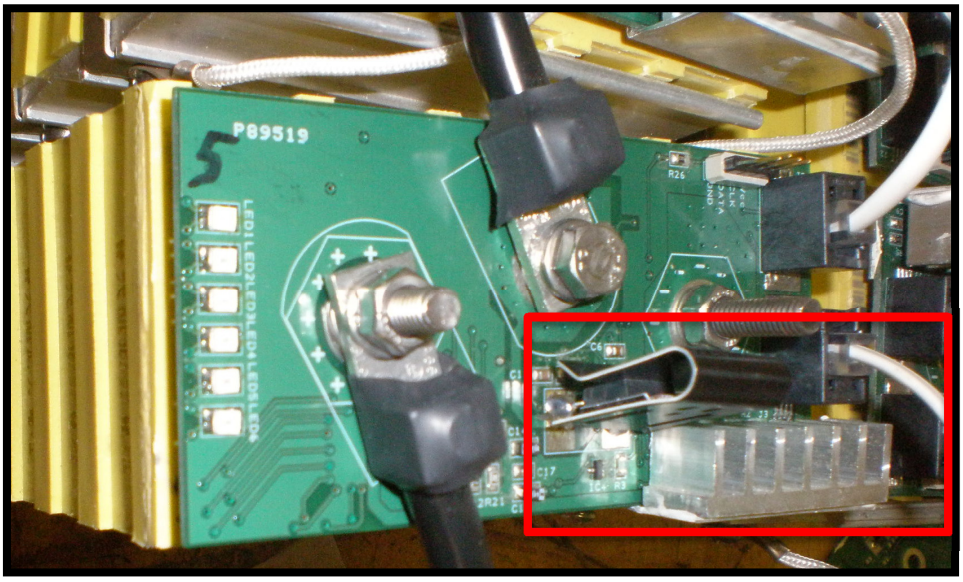

Figure 5.14: The red box shows the location of bypass resistor and MOSFET configured to be a voltage-controlled resistor (underneath silver heat-sink).

5.1.7 Implementation of Cell Temperature Monitoring

An ideal place to monitor the temperature of the cell is inside of the cell itself. However, since placing a sensor inside is not feasible for an end-user of a cell, the only option was to make an external measurement. The temperature sensor was placed near one of the cell's terminals. The reasoning for this placement was that it was assumed that the thermal transfer from inside the cell is faster through the cell's terminals and not through the plastic casing of the cell. The choice of which terminal to place the temperature sensor was chosen arbitrarily. The actual temperature sensor and location on the ICMU board is shown in Figure 5.15. A more accurate linear voltage output temperature sensor was utilized, namely the LM20. The circuit utilized is shown in Figure 5.16. The only high temperatures observed during battery pack use were while the board was bypassing current and dissipating heat through the bypass circuitry. This implies the board design is such that the heat distributed from the bypass circuitry influences the cell temperature measurement. This could be considered and possibly corrected for in software but for a future design, ideally better heat sinking of the bypass circuitry could decrease the influence on cell temperature measurements.

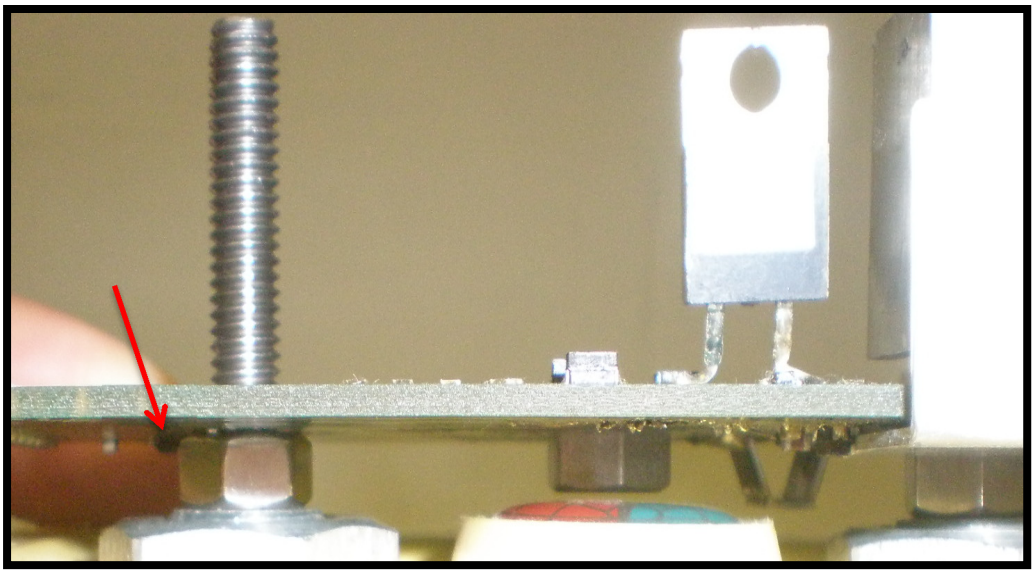

Figure 5.15: Location of cell temperature sensor near cell terminal bolt (small black integrated circuit identified by arrow).

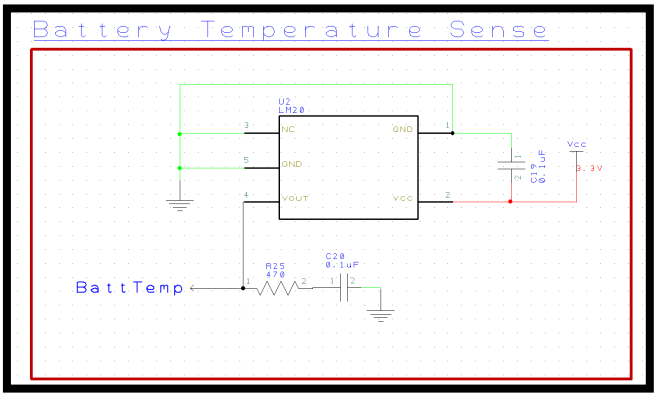

Figure 5.16: Schematic: Temperature sensor to measure cell temperature on ICMU PCB.

5.1.8 Implementation of LED displays for Status Indication

LEDs were placed onto each ICMU board to provide feedback information on the current status of the cell being managed by the ICMU as well as the battery pack as a whole. As pictured in the schematic of Figure 5.17, six dual colored LEDs were provided per each

ICMU. The actual implementation of a single ICMU can be seen in Figure 5.18. Although six dual colored LEDs were not necessary for display purposes, during the design process, the microcontroller had additional I/O open and so the number six was arbitrarily chosen for the number of dual LEDs. For display purposes only, when all cells were lined up, a "Knight Rider" (1982 TV Series) light effect was displayed. The BPMU broadcasted a command to turn on specific LEDs on each ICMU's individual LEDs. The commands were sent in such a way that it gave the appearance of a single lit LED traversing through all ICMUs from ICMU to ICMU. This provided a visual as to whether or not the communication between each ICMU and the BPMU was active.

A single dual LED, PD0 and PD1 (see Figure 5.17), was used to show the status of the onboard current bypass. Green indicated no bypass was active, Amber indicated that some current bypass was active, and Red indicated the maximum bypass current was being shunted around the cell.

In the event of an overtemperature, overvoltage, or undervoltage condition, all ICMUs were told to flash all 6 of their LEDs red by the BPMU. This was simply used to provide a visual for the battery pack user so that proper action could be performed (i.e. stop using the battery pack, or remove charger and end the charging process).

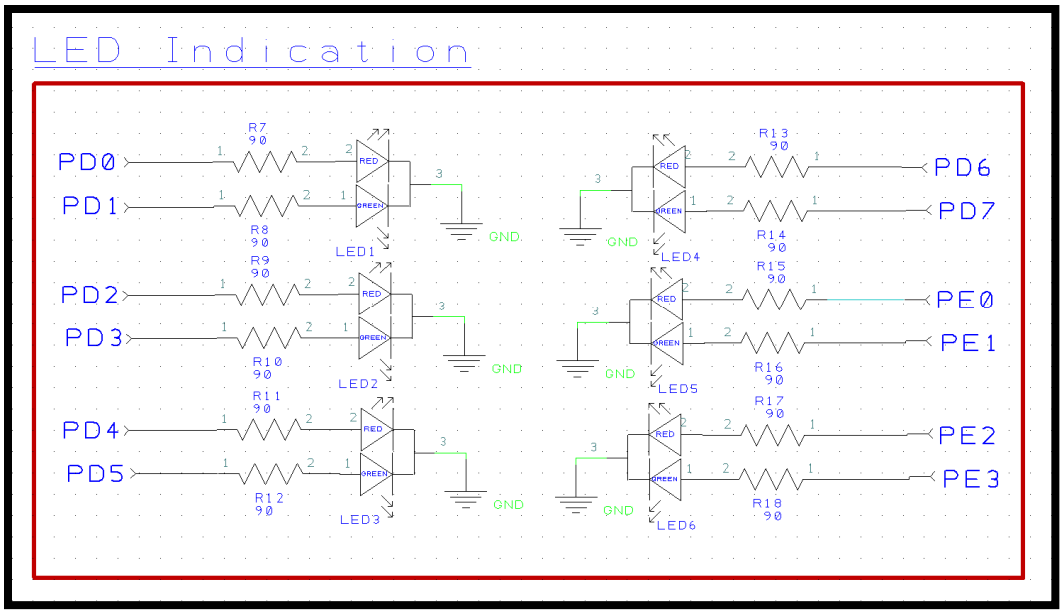

Figure 5.17: Schematic: Status LEDs on ICMU PCB.

Figure 5.18: Actual Implementation of dual color LEDs.

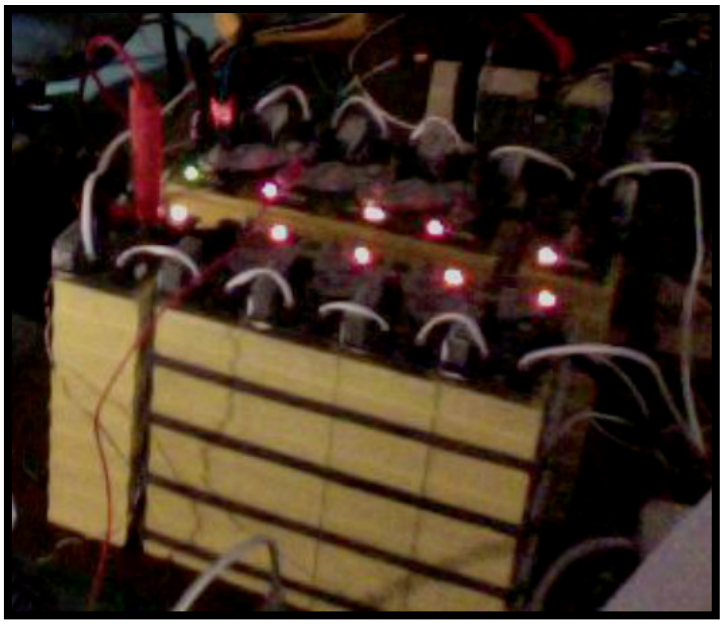

Figure 5.19: Battery Pack with ICMUs displaying various LED colors to demonstrate cell condition.

5.1.9 Implementation of Communication Circuits between BPMU and ICMU

In order to directly communicate from ICMU to the BPMU, isolation circuitry was required. This was done mainly because each cell was connected in series and so inherently the communication signal voltage is level shifted. For example between the lowest cell and highest cell within a series string, a serial digital output on the highest cell could have been as high as +30V higher than a serial digital output on the lowest cell. Therefore simply connecting the digital grounds between the highest cell and lowest cell was not possible as that essentially is a direct short from +30V to the pack ground. This is why the ADUM1250 I2C isolated digital bus IC was utilized. I2C is a serial communication protocol that is implemented with a Clock line, a bi-directional Data Line, and a Ground. The clock line and data lines are typically pulled high via pull-up resistors to the digital bus voltage. Therefore the clock and data lines get simply pulled down to assert an active signal. The

ADUM1250 simply allows the I2C communication to be done through specialized optical-isolated circuitry that allows bi-directional data on a single line as well as a bi-directional drive for the clock line as well. On the ICMUs on one side of the isolation barrier the ADUM1250 is supplied power from the cell the ICMU is attached to. On the other side of the isolation barrier the ADUM1250 is supplied power via the BPMU. To make connections from the ICMU to BPMU, a daisy-chain style connection is utilized on each ICMU. There are two paralleled RJ-11 4-PIN telephone jack style to achieve this. The circuit and actual implementation for this is pictured in Figure 5.20 and Figure 5.21, respectively.

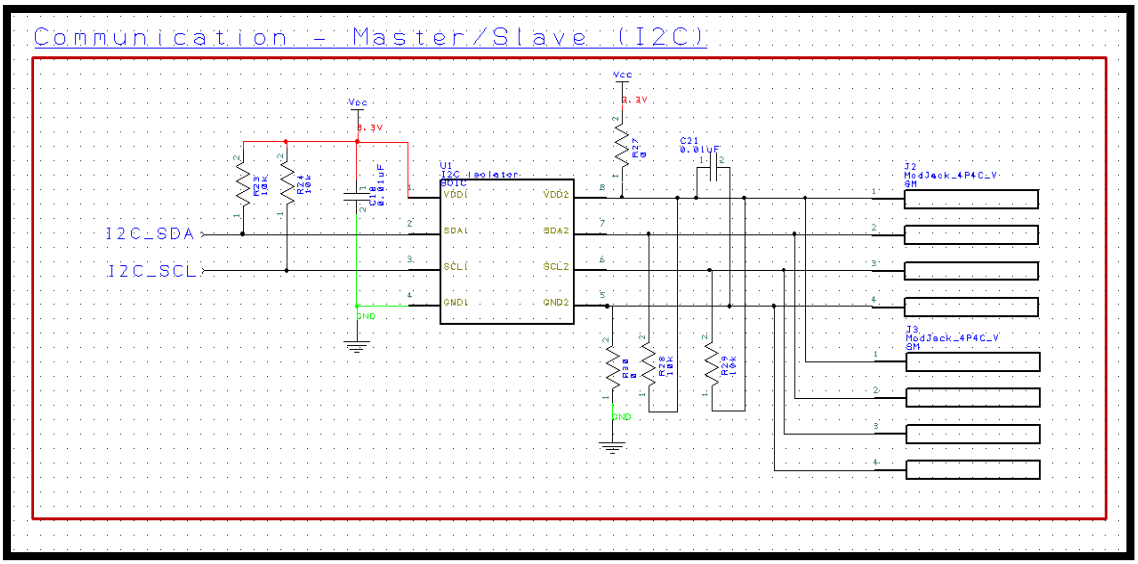

Figure 5.20: Schematic: Isolated communication I2C Bus on ICMU PCB.

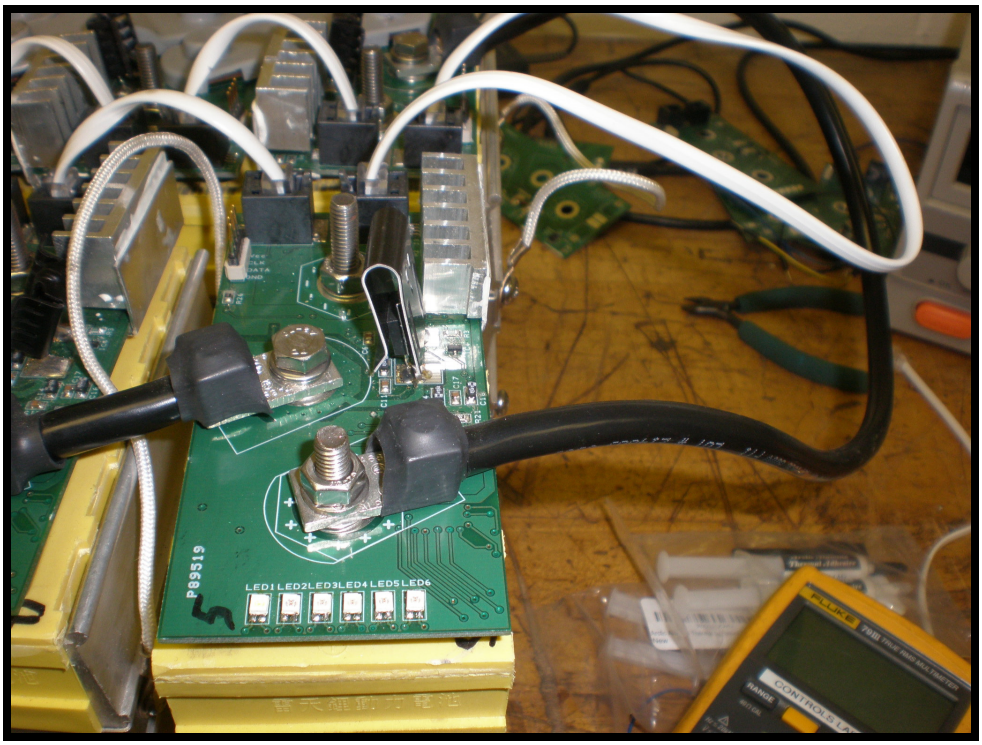

Figure 5.21: I2C connections shown from daisy chaining ICMU to ICMU.

5.1.10 Implementation of ICMU Address Hardware Identification

Each ICMU node was provided an identifying address that was provided with a 4-bit value that was determined by the value of 4 available digital input pins on the processor. The initial envisioned method to assign each ICMU's address was to simply program a unique address in each devices' memory, however, there were difficulties encountered with this method. Since the decision was made after the PCB was designed and made, the idea of taking 4 available pins on the processor was considered. By simply soldering each of these 4 pins either high or low (either shorting to a nearby low-ground or high-microchip operating voltage) each board could be assigned its own unique address with various combinations on these pins. Although up to 16 possible addresses could be used, only 10 were required for all ICMUs.

Upon initialization, each ICMU read these 4 pins to determine its own address. After determining its own address, the ICMU flashed a LED the value of its address to provide visual indication of the address value. Using these addresses the BPMU could individually poll each of the ICMU nodes over the I2C bus.

5.1.11 Hardware for BPMU/ICMU Hybrid

The hardware for the BPMU/ICMU are exactly the same since the BPMU/ICMU hybrid must be able to perform all the functionality of just a single ICMU board. The only differences are the BPMU has special connections for the I2C interface and the BPMU has an extra serial port output to communicate to the charger device. Figure 5.22 demonstrates the different connections used for BPMU. On the BPMU, instead of using the I2C isolator ADUM1250 integrated chip, the BPMU simply shorts the I2C data and clock lines pins 2 to 7, and pins 3 to 4, respectively. Zero ohm resistors are populated on R27 and R28 to connect the isolated side of the I2C Bus on all ICMUs to the BPMU's operating voltage Vcc and Gnd, respectively. In other words, the BPMU powers the right hand side of all the ADUM1250 devices on each ICMU. As for the left side of each ICMU's ADUM1250, each ICMU powers this side on its own.

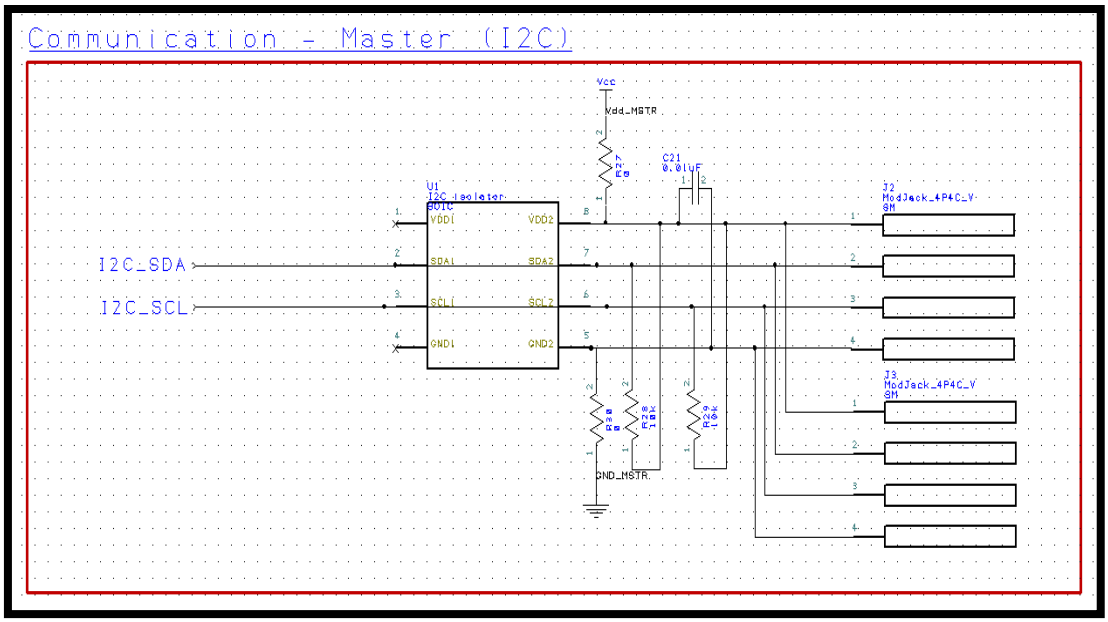

Figure 5.22: I2C connections for BPMU.

CHAPTER VI

SOFTWARE FOR BMS IMPLEMENTATION

This chapter discusses the software implemented for the integrated charger and BMS platform. Since 1 of the 10 ICMUs served both as an ICMU and the BMPU, two sets of software were written. The first set was used to control the 9 ICMU slave boards and the second set was for the ICMU/BPMU Hybrid.

6.1 ICMU Software Overview

During operation, a high-level view of the operations that each ICMU performs every 4ms is listed below:

1. Read Analog Values
2. Filter, Convert, and Process Analog Readings
3. Run control loop for Bypass
4. Package ICMU's Data into Communication Packet for BPMU
5. Update Status LEDs

6.1.1 Analog Voltage Reading

The analog-to-digital converter (ADC) measurement is performed with the XMEGA A4 processor's onboard 12-bit measurement. An interrupt is setup that rapidly and automatically sweeps through the four desired measurements. In the ADC interrupt the

four desired measurements are over-sampled with 16 fast samples and averaged to reduce sampling noise. The XMEGA A4 processor also has a programmable gain amplifier block that can be used to amplify the signal (either differentially or single-ended) before being sampled by the ADC. The available gain selections are 1, 8, 16, and 64. The cell voltage measurement is performed with gain of 1 on the programmable gain amplifier. The cell current measurement is performed differentially with a gain of 8. The bypass current measurement is performed differentially and also with a gain of 1. Lastly, the temperature measurement is performed single-ended with a gain of 1.

6.1.2 Read, Filter, Convert, and Process Analog Readings

After each analog measurement is averaged from the 16, samples each is placed into its own 64-element ring buffer every 4ms. This buffer is used to calculate a 64-sample moving average to filter out any high frequency noise. Each value is converted from the raw ADC value to scientific units (e.g. volts, amps, degrees Fahrenheit, etc…). Each of these conversions used linear equations consisting of a gain and offset.

Using the converted data, cell voltages were checked to determine the state of a cell and to detect overdischarge (under 2.5V) or overcharge (larger than 4.2V). Also calculated was the integral of the current measurement. This provided a rough SoC estimate. The integral was performed simply by accumulating the sum of the currents multiplied by the sampling time of 4ms.

6.1.3 Run Control Loop for Bypass Current

After the measurements were taken, the cell voltage was checked to determine if it was higher than 3.8V. If it was, the bypass current controller was activated; otherwise this

section of code itself was skipped. However when it was activated, an integral control loop ran to determine how much current should be bypassed to keep the cell at a nominal 3.8V. The control loop's output was the duty-cycle value that was used to drive the low-pass filter coupled to the gate and source of the aforementioned bypass MOSFET of Figure 5.13. The PWM peripheral used to control this duty-cycle was set-up for a frequency of approximately 20.8 kHz (chosen simply to ensure the frequency was not audible and higher than the low-pass RC filter). Provided that the voltage applied to the MOSFET's gate and source was slowly increased, this causes MOSFET's operating region to be placed into its linear region of which the MOSFET behaves as a voltage-controlled resistor. Once the controller increases the duty-cycle beyond the MOSFET's gate threshold, the MOSFET reaches its saturation region and becomes relatively low impedance compared to its linear region operation. At this point, the MOSFET's impedance becomes relatively negligible. The 2.1 ohm resistor connected in series with the MOSFET, is now the only current limiting impedance. As will be discussed later, the BPMU will need to take action once the maximum current is bypassed and the cell voltage can no longer be maintained at 3.8V.

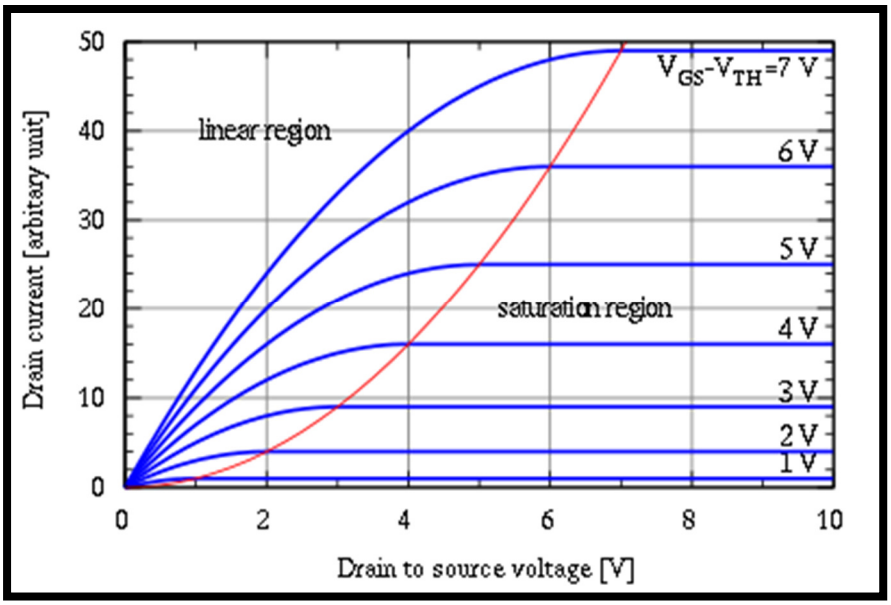

Figure 6.1: MOSFET operating regions based upon gate voltage, drain current and drain to source voltage.

Since the cell's dynamics were not rapidly changing, the tuning of the controller for high performance was not critical. The controller's integral gain was chosen through an offline manual iterative tuning process. This process consisted of starting with low gains and increasing them in small incremental steps. The final integral gain was chosen conservatively to prevent the bypass current value from overshooting and varying wildly. However, it was just enough speed to increase or decrease the bypass current to maintain the cell's 3.8V.

6.1.4 Package ICMU's Data into a Communication Packet for BPMU

Next, after the various measurements were sampled, calculated, and/or converted, it was placed into a fixed communication packet. Each ICMU stored this packet in anticipation of a data update request from the BPMU. The communication packet formed consisted of the Cell Current, Cell Voltage, Bypass Current, estimated SoC, and the

following status bits: bypass on/off, bypass is/is not at maximum duty-cycle, cell is/is not overdischarged, and temperature is/is not over maximum threshold. In terms of size, the packet consisted of a total of 10 bytes, 2 bytes for each of the 4 analog readings, and 2 additional bytes for the SoC and the 4 status bits.

6.1.5 Update Status LEDs

The last portion of the ICMU's operation loop was used to update the ICMU's dual color status LEDS. One of the six LEDs served as an indicator for the state of the ICMU's bypass circuit. The other five LEDs were used to indicate active communication with the BPMU as well as system status. Not all five dual color LEDs were necessary and were merely placed originally on the ICMU PCB for debugging purposes. However, since they were available on the board they were creatively utilized.

The single dual-color LED was used to provide the ICMU's bypass with the following color configuration: In order to indicate that the Bypass was off, the Green LED was individually lit. In order to indicate some current was being bypassed and less than the maximum, the Green and Red LEDs were lit (creating Amber). Lastly, if the maximum current of 1.9A was being shunted around the cell by the bypass circuit, the Red LED was individually lit.

All display LEDs were updated with values from the BPMU for the "Night Rider effect", and as aforementioned, this concept was used to display the battery pack status and also as a visual indication that the communication between the BPMU and all ICMUs was active.

6.2 ICMU Interrupts

There were three interrupts that ran as needed over the ICMU's operation loop, the I2C Interface Interrupt, Timer Interrupt, and the ADC Interrupt. Each of these will be discussed in the following sections.

6.2.1 I2C Interrupt

The I2C Interrupt was used for communication between the BPMU and each ICMU. The BPMU/ICMU hybrid and the other 9 ICMUs were all placed onto the same I2C bus via the I2C isolation chips mentioned in the hardware design section. Each ICMU's I2C peripheral interface was setup as a slave module with the address determined by the four address selection pins as aforementioned. The ICMU then only responds to I2C requests provided with its matching address from the BPMU. The BPMU performed two operations in regards to the I2C interface; a data packet update read sequence or a communication status write sequence.

The read sequence simply consisted of the BPMU commanding data to be given from a specific addressed ICMU. Upon the request and verification of the address match the I2C interrupt was triggered. Inside the interrupt the slave firsts acknowledges the BPMU's request for a packet update indicating it had at least one byte of data to send. Next, the slave starts providing the packet data byte by byte and continues to acknowledge that it had data to send. When the ICMU reached the end of its 10-byte communication data packet, it simply ignores the acknowledgement and the BPMU stops the I2C read sequence.

During the write sequence the BPMU broadcasts a message to all ICMUs. This message was a number that corresponded to one of the 60 Green LEDs available from each ICMU. This number was updated every time the BPMU finished a read sequence with an ICMU. The pattern the BPMU followed to update the number consisted of incrementing the number from 1 to 60 and then decrementing from 60 back down to 1. This is what provided the "Night Rider Effect" as mentioned earlier. Since this message was broadcasted to all ICMUs, provided the ICMU's green LED corresponded with the broadcasted value, it turned on that specific LED. This visually provided a means to see when a message request was dropped if the "Night Rider Effect" pattern was not consistent (i.e. if the LED stopped on a single LED for inconsistent period of time from the others, or if LEDs were skipped in the pattern, etc…). In addition to providing communication status using the write sequence, the BPMU also used this write sequence to provide all ICMUs with a message to blink their RED LEDs. This was reserved for the event when the BPMU processed the data it received from all ICMUs and detected a cell with an overcharged, overdischarged, or overtemperature condition. This provided a visual indicator for the user to be able to take action with the system.

6.2.2 Timer Interrupt

This interrupt was setup to occur approximately every 4ms. The only purpose for this timer was simply to perform the integration of the current for the SoC integration.

6.2.3 ADC Interrupt

The ADC was setup to read a sweep of the four ADC measurements mentioned previously. The interrupt occurred at the completion of the fourth ADC conversion. In the interrupt, each individual conversion was summed up over 16 samples and then averaged.

It is this 16 sample average that was used each time the ICMU operation loop was run. This 16 sample average was implemented to reduce sampling noise.

6.3 BPMU/ICMU Hybrid Software Overview

Since the BPMU is actually a hybrid BPMU/ICMU in this BMS implementation, the BPMU performs the same functions of the ICMU. Therefore, the discussion of software for the BPMU/ICMU hybrid will only focus on the differences between the code sets used to implement the stand-alone ICMU versus the code for the BPMU/ICMU hybrid.

In terms in the functionality of the ICMU portion of the BPMU/ICMU hybrid, there is only one main difference, the I2C communication. The BPMU/ICMU hybrid does not send any information out on the I2C bus about the cell it is managing like the other ICMUs. It simply just stores its own information internally and then it gathers the rest of the battery pack's cell information from other ICMUs via the I2C bus. All other steps mentioned in the ICMU software overview behave the same in the hybrid device. The operation for the BPMU portion of the BPMU/ICMU hybrid consist of the following: All steps for ICMU portion are executed first. Next data from ICMUs are processed, converted, and printed out. Next, cell status is determined and any necessary actions are taken. Lastly, charger detection, charge current selection logic, and charger communication is performed.

6.3.1 Process ICMU Data Packets, Convert and Print out

After the BPMU requests data from a particular ICMU on the I2C bus, an interrupt processes all the data received (this interrupt will be discussed further in a later section). Once an entire packet is received from the ICMU a flag is set indicating there is new data to be processed. The BPMU takes the data from the packet and decodes it back to the

individual measurements and status bits cell current, cell voltage, cell's bypass current, cell's estimates SoC, temperature, bypass status bit, bypass at max bit, overtemperature detection bit, and the overdischarge detection bit. This data is then sent out from a BPMU's information serial port at a baud rate of 19200. This process is repeated until all information is received from all of the other 9 ICMUs.

The collection of all data from each ICMU occurs in 100 milliseconds. This means that the acquisition of data from all ICMUs is complete once every second. When a full round of data is received from all ICMUs, the BPMU then calculates the overall pack voltage, current and the SoC. The pack voltage is determined by summing the individual cell voltage measurements received. The pack current is determined by taking an average of all the cell currents measured (since all cells are connected in series). The SoC is simply given as the lowest SoC of the lowest charged cell (again since all cells are in series, to protect the lowest charged cell from overdischarge, the lowest SoC percentage is given). Next, all ICMU data is analyzed for various conditions and action is taken as necessary.

6.3.2 Determine Cell Status and Take Corrective Action as Necessary

The BPMU takes all the data and analyzes each ICMU to see if any of the following conditions: undervoltage, overvoltage, overtemperature, and bypass at max current.

To detect the overdischarge condition, the overdischarge status bit sent from each ICMU is evaluated. If this bit is received with a positive for an overdischarge condition for two consecutive messages for the same ICMU, action is taken. Since there is no load shut-off in this BMS implementation, the only action that is taken is to indicate to the user that

the condition exists. To do so, the BPMU simply tells all ICMUs to flash their red LEDs in sync. This leaves it up to the user to stop the use of battery pack.

In order to detect an overvoltage condition, the BPMU evaluates the level of each ICMU's cell voltage during the charging process. If the BPMU detects that any cell's voltage is larger than 4.2V after 2 seconds, the BPMU sends a command to charger telling it to shut-off.

As for the overtemperature condition, again the BPMU simply scans each individual ICMU's overtemperature status bit. If any device stays in the overtemperature condition over 2 seconds, the BPMU sends a command to the charger telling it to also shut-off in this case.

Given that none of the aforementioned condition exist the charging process runs until all cells are charged equally as will be discussed in the following section.

6.3.3 Implementation of I2C Communication Scheme

The BPMU polled one ICMU node every 10ms. In other words, to poll and then receive all 10 ICMU nodes' cell information, this required a total of 100ms. This sampling time was chosen to ensure any fast transient current pulses were captured during discharge.

6.3.4 Implementation of Charger Control Interface

The charging device used to implement the proposed method was developed in conjunction with an electric vehicle project (Taschner, 2011). Shown in Figure 6.2, is a high-level schematic of the charging device. It consists of a bridge rectifier, a smoothing capacitor, and a buck-regulator block. Not pictured are additional electronics that provide the monitoring, measurements, controls, and serial interface for the charging device. An

actual photo of the hardware is shown in Figure 6.3. Next, the serial interface will be discussed in more detail.

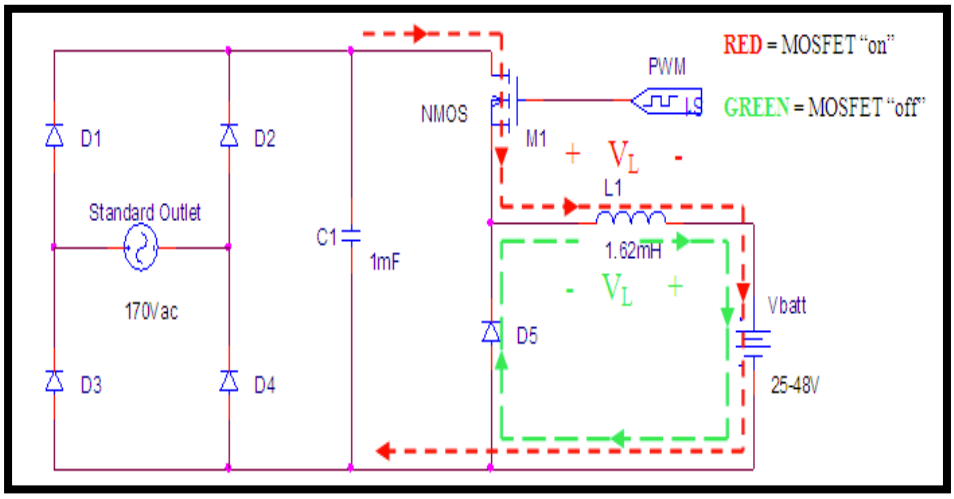

Figure 6.2: A high-level schematic of the charging device (Taschner, 2011).

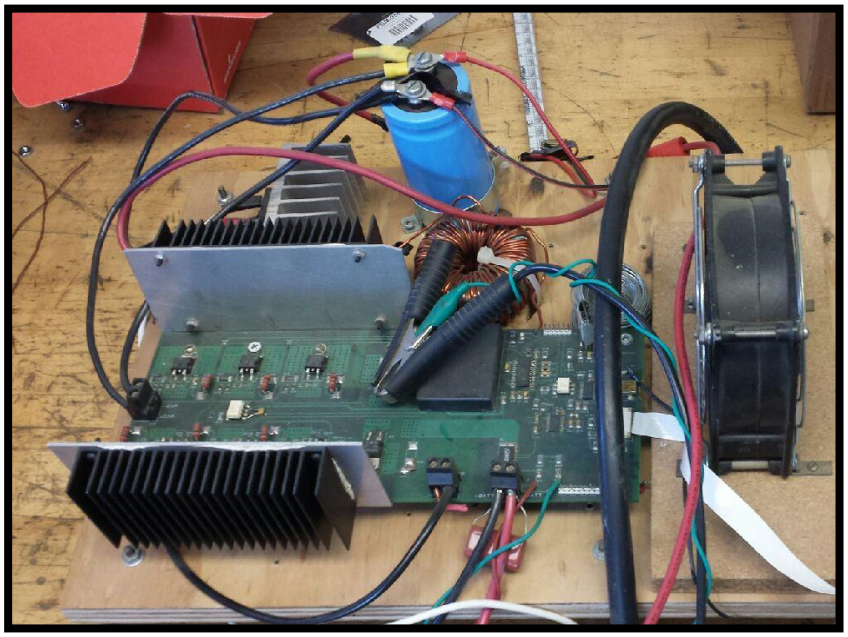

Figure 6.3: Photo of charging device (Taschner, 2011).

The serial communication interface between the BMS and the charger consisted of three wires. Two of these were signals and the last wire was used to establish a common ground. One of the signal wires was used to detect whether or not the charger was presently connected to the BPMU. The other signal wire allowed the BPMU to transmit commands to the charger.

In order to detect the presence of the charger, the BPMU used a digital input with a high impedance pull-up resistor. When the three-wire connection was made between the charger and the BPMU, the digital input was pulled to ground.

The command packet transmitted by the BPMU to the charger consisted of 6 bytes as summarized in Table 6.2. This packet was designed to transmit a new desired charging current from the BPMU to the charger. Error detection capabilities were built into the packet. The desired current will have two integer digits and two decimal digits. A variable representation of this format can be viewed as "II.DD".

The starting character (Byte 0) and an ending character (Byte 5) envelops the packet to allow for synchronization of the messages between the BPMU and charger. The integer portion (Bytes 1 and 3) and decimal portion (Bytes 2 and 4) of the current is redundantly sent within the packet. This is to allow the charger to receive both sets of bytes and check if any bits were changed during the transmission. To provide an additional measure of error checking, the BPMU sends this entire packet 5 times repeatedly to the charger. The charger then ensures all packets are consistent before updating the actual charging current to the new desired value.

Table 6.2: Communication packet sent from the BPMU to the charger.

BYTE #	**DATA**	**DESCRIPTION**
0	0xAA	Starting character
1	II	Integer Portion of Current
2	DD	Decimal Portion of Current
3	II	Integer Portion of Current
4	DD	Decimal Portion of Current
5	0x55	Ending character

CHAPTER VII

SIMULATION VALIDATION AND RESULTS

This chapter will discuss the results obtained from a battery pack discharge and charge cycle with the proposed integrated charger and BMS implementation. Plots of the data collected during both cycles will be shown. In addition, some images collected of the BMS system during the charging process will be displayed.

7.1 Discharge and Charge Cycles Results from a 4-cell Battery

The positive results from the simulations of the integrated BMS and Charger of Section 4.2.3 were used to perform a quick manually-controlled experiment on a series connected 4-cell battery pack. The cells consisted of four 26650 $LiFePO_4$ cells, each with a 2.5Ah capacity. A power supply was used for the charging cycle and a programmable load was used for the discharge. The power supply was set for a constant-current of 4A and a constant-voltage of 14.8V (3.7V x 4 cells). The programmable load on the other hand was set for a fixed discharge of 4A. On each cell a resistive bypass was implemented consisting of an 8.2 ohm resistor that turned on at 3.5V.

The experiment started with all cells individually charged with the same CC and CV profile. For the constant-current mode 4A continuous was maintained until the cell voltage reached 3.7V. In the constant-voltage mode, the cell voltage of 3.7V was

maintained until the cell reached 50mA. At this point the charging was complete and the cells were allowed to rest.

The purpose of the experiment was to demonstrate that the cell balancing could be more effective if the charging current was controllable. This could be accomplished by manually watching the cell voltages and reducing the power supplies charging current as necessary.

The experiment started with cycling the pre-balanced cells for two cycles. As can be seen in Figure 7.1, the first two cycles show that the cell voltages stayed relatively balanced during the charge cycle (note that the second cycle appears different after the charge cycle only because the pack was allowed to rest longer). After the second discharge cycle, a single cell was individually charged at 4A on its own for 10 minutes to purposefully unbalance the pack (unbalance marker in yellow in figure). After this, the cells were allowed to rest again and then the third charge cycle was started. As can be seen, the higher charged cell quickly reached 3.7V before the other cells. Even with the bypass on, the cell voltage increased significantly more as observed in the simulations of Chapter 4. At this point, the charger current was turned down to a constant-current of 0.4A. This allowed the bypass to completely shunt the current around the highest charged cell while the other cells remained charging. Once all cells reached the bypass activation voltage, the current was increased to 0.6A to allow all the cells to continue charging. The next charge and discharge cycle show that the charging stays well balanced. After a long rest, an additional charge cycle was performed to demonstrate the cells remained balanced.

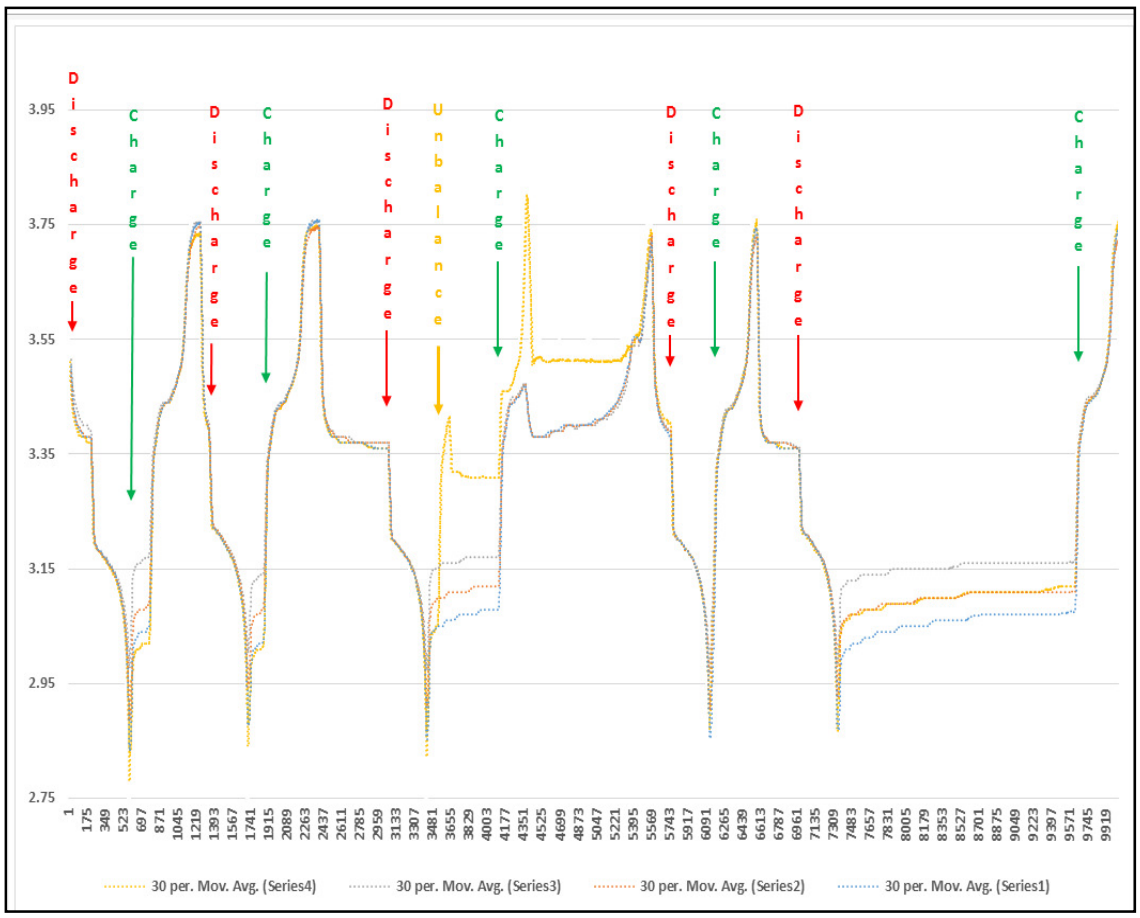

Figure 7.1: Data plotted from a 4-Cell series-connected battery pack manually-controlled experiment.

7.2 Results from the Integrated Charger and BMS on a 10-Cell Battery Pack

The results of data collected from the 10-Cell series-connect integrated Charger is given in the next few sections. The data collected from the discharge cycle is provided followed by the charging and balancing results.

7.2.1 Discharge Cycle

A fixed high power resistive load with a resistance of approximately 1.67 ohms was used to discharge the battery pack at a nominal discharge current of 20A. The discharge

was stopped when at least one of the cells dropped below the 2.5V threshold. The BPMU visually indicated this to the users by notifying the ICMU boards to flash their red indicator LEDs. As can be seen in Figure 7.2, Cell 10 reached a low voltage of 2.5V. At this point the red LED indicators were asserted and removal of the load completed the discharge cycle. Also worth noting is that the cell voltages have different discharge characteristics. As can be seen in the figure, initially in the discharge cycle cell 4 is at a lower voltage. However, towards the end of the discharge, cell 10 reaches 2.5V first.

Also shown at the bottom of Figure 7.2 was the discharge current. Since the resistance was a fixed resistive load, the cell discharge current started out higher as the cell voltages were higher earlier on during the discharge cycle. The linear cell voltage region, where the cells stays relatively constant, can be seen from approximately 0.5 to the 2 hour mark of the discharge cycle. Towards the end of the discharge cycle (2.5 hours and after), the cell voltages move out of the linear region and began to drop steeply. As can be observed and expected, a decrease in discharge current occurs. At the end of the cycle, the removal of the load is seen and the measured cell discharge current drops to zero.

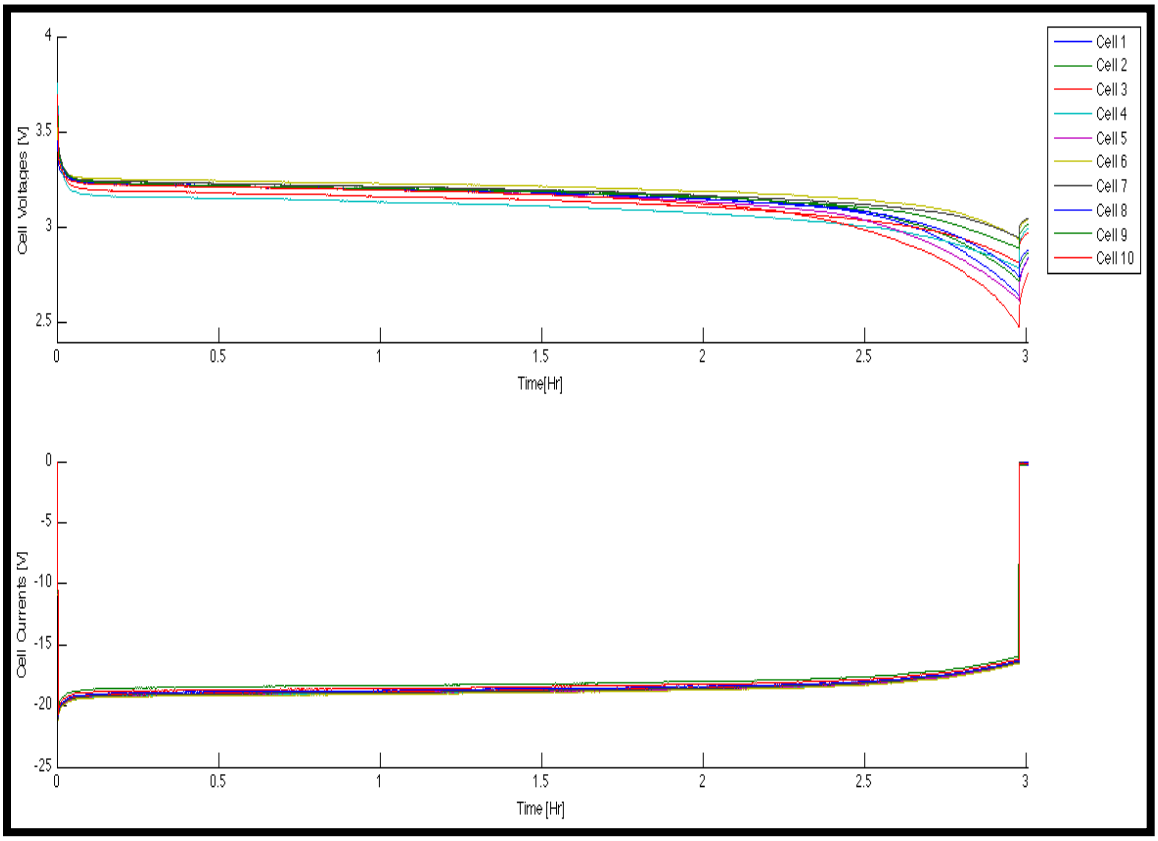

Figure 7.2: Discharge cycle of 10 Cell Series Connected $LiFePO_4$ Battery Pack.

7.2.2 Charge Cycle

This charge cycle occurred after the aforementioned discharge cycle was completed. Once the BPMU was connected to the charger, it communicated to the charger and requested a charging current of 12A. After the start button was pressed on the charger, the BPMU continuously polled each of the ICMUs for information about their respective cells. Figure 7.3 shows the visual LED feedback from each ICMU early on during the charging process. Since no cells are in bypass all feedback LEDs are green.

Figure 7.3: Battery Pack with ICMU and ICMU/BMPU Hybrid attached to each cell. As pictured here each cell is charging and is not bypassing any current.

The ICMUs on each board watch over its own cell and begin bypassing current around the cell if necessary. When an ICMU begins bypassing current, the feedback LED on the ICMU changes to amber. When an ICMU begins bypassing current close to its maximum allowable current of 2A, the feedback LED changes to RED. Figure 7.4 shows the system later in the discharge cycle with some ICMUs in bypass. As can be seen, some LEDs are currently in bypass and amber. One LED is red since that bypass has reached the maximum shunt current and can no longer keep the cell at the desired cut-off voltage. At this point, the BPMU will communicate to the charger that it needs to reduce its charging current by 2A.

Figure 7.4: Battery Pack with ICMU and ICMU/BPMU Hybrid attached to each cell. As pictured here, 4 cells are bypassing current since they have reached a voltage of at least 3.8V.

This process continues until all cells have reached the same charge cut-off voltage. Figure 7.5 shows the battery pack late in the charging cycle. As can be seen, all but one cell is bypassing current except the ICMU/BPMU hybrid. This is because the ICMU/BPMU hybrid powers the isolated I2C bus for all of the other ICMU cells. This is in addition to the additional processing it must perform than the other ICMU cells. The last figure of interest, Figure 7.6, shows the battery pack moments before it discontinues the charging process as the ICMU/BPMU hybrid gets to the desired charge cut-off voltage.

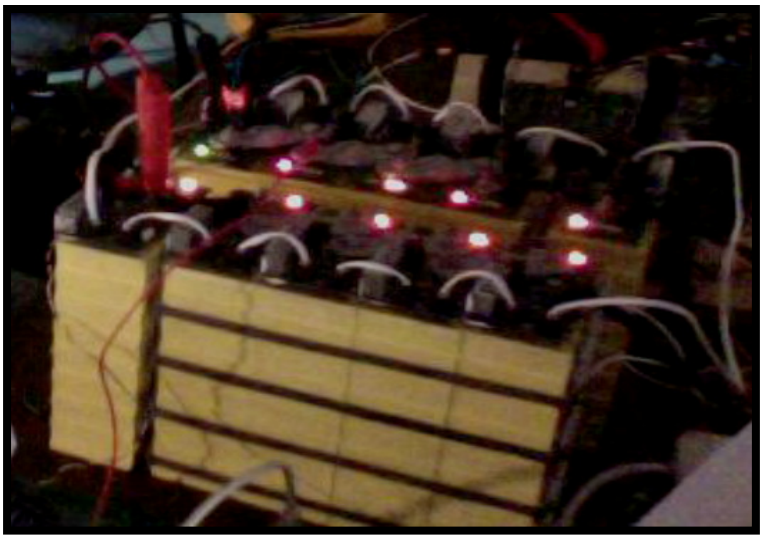

Figure 7.5: Battery Pack with ICMU and ICMU/BPMU Hybrid attached to each cell. As pictured here, all cells are in bypass with the exception of the ICMU/BPMU Hybrid.

Figure 7.6: Battery Pack with ICMU and ICMU/BPMU Hybrid attached to each cell. As pictured here, all cells are in bypass. Shortly after this picture was taken, the BPMU commanded the Charger to discontinue the charging process.

The data collected from the charging cycle is shown in Figure 7.7. Pictured in the figure are each of the individual cells' voltage, the charging current, and each cell's bypass

current as measured by each cells' ICMU board. During the initial phase of the charge cycle, a wide range of cell voltages was present after the discharge. However, as the charging commenced, the cell voltages converged when they each reached the nominal voltage plateau of 3.3V. However, as the cells neared their full charge capacity, the cell voltages began to slightly diverge. However, the BPMU and ICMU circuitry mitigated damaging cell conditions by cooperating to ensure that no cells reached an overvoltage state.

As can be seen in the bypass current measurement plots, once a cell reached 3.8V, its ICMU activated the bypass circuitry for that given cell. The bypass circuitry continued to bypass the necessary amount of current in order to maintain the cell's voltage at 3.8V if possible. However, once the maximum amount of current was bypassed, the cell continued to charge. This increased the highest charged cell's voltage to the maximum voltage of 4.2V. At this point, the BPMU told the charger to decrease its charging current to 1.8A. At this charging current, the ICMUs individually control their cell's charging current by controlling the bypass circuitry. Once all cells reached their full bypass current, the BPMU requested the charger to end the charging process.

In summary, the method of modifying charging current during balancing allows cells to balance in fewer cycles. In fact, the manually-controlled experiment from Section 7.1 used this method and demonstrated the battery pack's cell voltages converging within a single cycle. The automated implementation of this method also demonstrated similar results. It can be seen that this balancing cycle resembles the simulated balancing cycle of Figure 4.15. Following the balancing cycle of Figure 7.7, the cells in the pack became balanced similar to those shown in the simulations of Figure 4.14.

7.3 Chapter Summary

This chapter provided results from a quick manually-controlled experiment, to demonstrate the benefit of providing an interface to the charger from the BMS. This allows the cost-effective method of resistive bypassing to be implemented at a faster rate as opposed to over a period of several charge/discharge cycles. Also discussed in this chapter were the results from the actual implementation of the proposed BMS with the charger interface. This implementation automates the same concept performed in the manually-controlled experiment. The results obtained were in-line with the methods outlined in the performed simulations. The results demonstrated the ability to charge and balance a relatively unbalanced pack in less cycles than using an independent charger and independent BMS with bypasses alone.

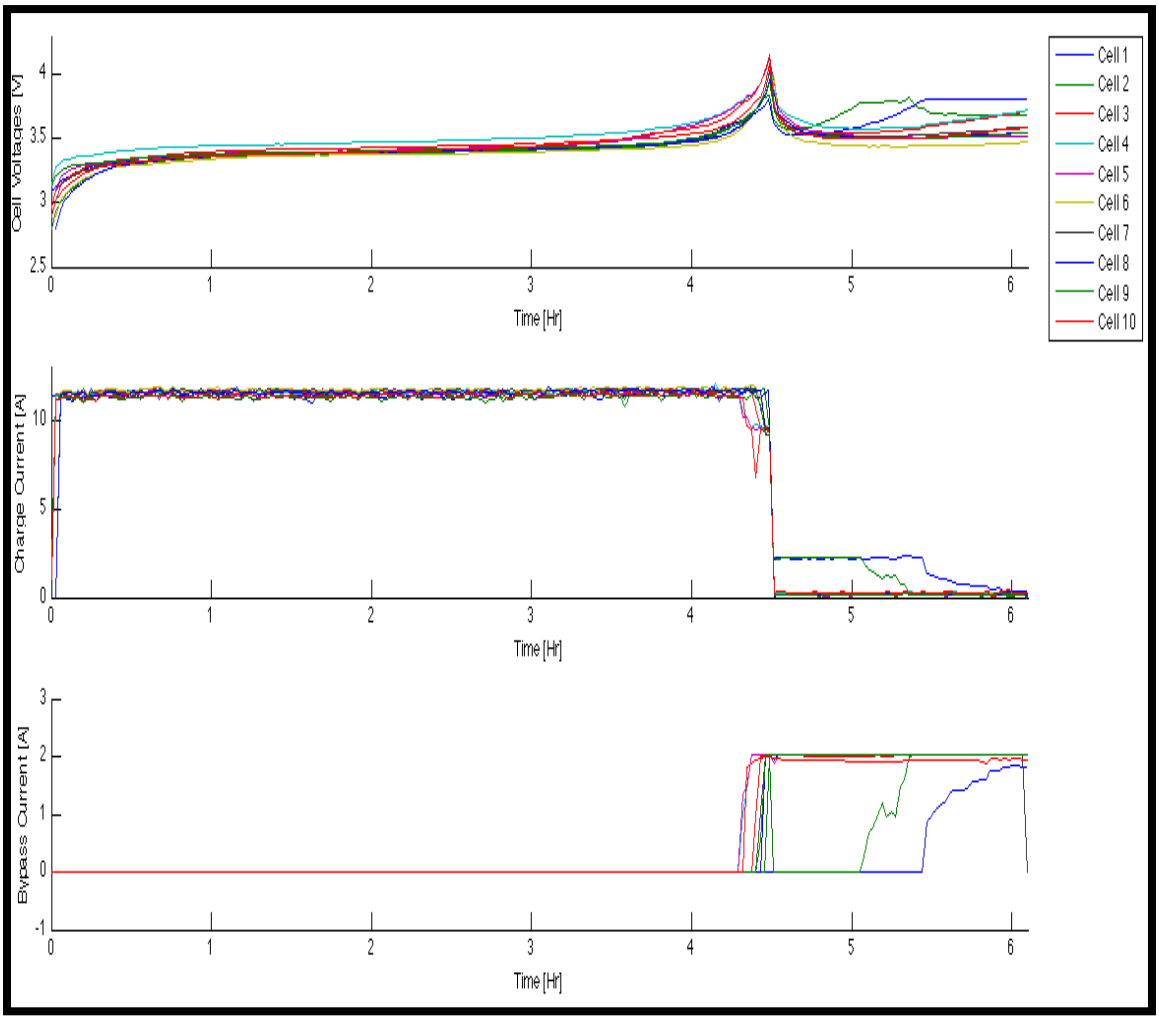

Figure 7.7: Charge cycle with integrated charger and BMS for a 10 cell series-connected battery pack.

CHAPTER VIII

CONCLUSIONS

In this thesis, and investigation into battery management systems and monitoring systems was performed. The investigation looked at the available methods of balancing cells within a battery pack and identified a cost-effective method, namely, the lossy passive bypass. It was proposed, to improve this method, to integrate the BMS with the charging device. Next, a summary of the process followed to simulate and test this method is provided along with the results.

Before being able to test the proposed method, a first-principles cell model was obtained. This was done by following a procedure that allowed the required cell dynamics to be observed. Following this procedure, the collected data was analyzed and additional data sets were made from additional calculations. A combination of the collected data set and new calculations data set were used to obtain the cell model's unknown parameters. This was done by performing batch least squares on the data sets.

Next, using this cell model and the parameters obtained, battery pack simulations were performed. These simulations were performed to understand how cells in a series-connected battery pack behave under various configurations. The main battery pack configurations tested were a: 1) battery pack with no cell balancing, 2) battery pack with

resistive bypasses and a BMS, and 3) a battery pack with resistive bypasses, BMS, and integrated charger.

The battery packs with no balancing had cells where their voltages diverged as the cells went through several charge and discharge cycles. In a practical application, the cells in this battery pack would get damaged by straying from their recommended manufacturing voltage operating range.

The battery pack with balancing circuitry on each cell had better results in terms of maintaining the desired voltage operating range. The cell voltages in this configuration converged with one another as the pack was cycled. Depending on how poorly balanced each cell was with respect to the others, the initial charge and discharge cycles could also force the cells to work outside the desired operating voltage range. However, as the charge and discharge cycles continued, the cells eventually became balanced.

The last configuration simulated, Configuration 3, had the best results in terms of the number of cycles it took to balance the battery pack. It allowed the cells to be balanced in a single cycle which prevented the cells from going outside the desired voltage range in the subsequent charge cycles. In order to demonstrate a physical implementation of this concept, hardware and software was developed.

The actual battery pack used for the demonstration was comprised of ten 60Ah $LiFePO_4$ cells connected in series. Circuit boards with a microcontroller and additional supporting electronics were designed to directly connect to each cell individually. Using the microcontroller and supporting electronics, code was written to monitor the cell's voltage, current, and temperature. In addition, the circuit could control bypass current

around the cell up to a maximum of 2A. Furthermore, serial communication circuitry allowed each cell to communicate information as well as receive commands from a single master device. This master device processed information from each cell and performed necessary actions to protect the battery pack. During a discharge cycle, the master provided visual feedback to a user if one or more cells reached a low SoC. During a charge cycle, the master interfaced to an integrated charger. The interface between the two devices allowed the master to change the charger's charging current in response to the battery pack's condition.

Lastly, results were provided from a manual experiment to demonstrate the balancing of the battery pack with the ability to change the charging current as necessary. In addition, data was collected from a discharge and charge cycle using the designed software and hardware that automated the concepts demonstrated in the manual experiment. It was demonstrated that an integrated BMS and charger could balance a battery pack in as little as a one cycle, thus providing a significant improvement over a non-integrated BMS-charter combination.

BIBLIOGRAPHY

Andrea, D., "Battery Management Systems for Large Lithium-Ion Battery Packs", ISBN 1608071049, September 2010.

Anders, A. "Tracking down the origin of arc plasma Science-II. Early continuous discharges," *Plasma Science, IEEE Transactions on*, *31*(5), 1060-1069, 2003.

Annavajjula, V. K., "A Failure Accommodating Battery Management System with Individual Cell Equalizers and State of Charge Observers," *A Thesis Presented to The Graduate Faculty of The University of Akron,* December 2007.

Bergveld, H. J., "*Battery Management Systems Design by Modelling,"* Royal Phillips Electronics. ISBN 90-7445-51-9, 2001.

Battery Space, "LiFePO4 Cells/Packs," *Powerizer,* Last accessed December 2010. <http://www.batteryspace.com>

Buchmann, I., and Cadex Electronics Inc., "Batteries in a portable world: a handbook on rechargeable batteries for non-engineers," *Cadex Electronics*, 2011.

Choi, S. S., and Lim, H.S., "Factors that affect cycle-life and possible degradation mechanisms of a Li-ion cell based on $LiCoO_2$," *Journal of Power Sources* 111.1, 130-136, 2002.

Daowd, M., Omar, N.; Bossche, V., et al., "Passive and Active Battery Balancing comparison based on MATLAB Simulation," *IEEE Vehicle Power and Propulsion Conference,* (VPPC 2011), pp. 1-7, 2011.

Economist, The, "*Case History: In search of the perfect battery",* Last accessed May 2014. < http://www.economist.com/node/10789409>. March 2008.

Krieger, E. M., Cannarella, J., & Arnold, C. B., *"A comparison of lead-acid and lithium-based battery behavior and capacity fade in off-grid renewable charging applications," Energy*, *60*, 492-500, 2013.

Hartley, T. T., Beale, G. O., Chicatelli, S. P., “Digital simulation of dynamic systems: a control theory approach,” *Prentice-Hall, Inc*., 1994.

Hartley, T. T., Jannette, A. G., “A First Principles Battery Model for the International Space Station,” *American Institute of Aeronautics and Astronautics,* AIAA-2005-5625, 2005.

Hartmann, R. L., “An Aging Model for Lithium-Ion Cells,” *A Dissertation Presented to The Graduate Faculty of the University of Akron,* December 2008.

Hicks, J. A., Gruich, R., Oldja, A., Myers, D., Hartley, T. T., Veillette, R., et al. “Ultracapacitor Energy Management and Controller Developments for a Series-Parallel 2-by-2 Hybrid Electric Vehicle,” *IEEE Vehicular Propulsion and Power Conference,* 2007.

ICCNexergy, “Comparison of Energy Densities for Various Battery Chemistries,” 2011. Last accessed March 2014. <http://www.iccnexergy.com/battery-systems/battery-energy-density-comparison>

Kalhammer, F.R., Kopf, B.M., Swan, D., Roan, V.P., Walsh, M.P., “Status and Prospects for Zero Emissions Vehicle Technology: Report of the ARB Independent Expert Panel 2007,” *Prepared for State of California Air Resources Board*, Sacramento, California, April 2007.

Kumar, R. Vasant, and Sarakonsri, T., "Introduction to Electrochemical Cells." *High Energy Density Lithium Batteries: Materials, Engineering, Applications,* 2010.

Moore, S. W., Schneider, P.J. “A Review of Cell Equalization Methods for Lithium Ion and Lithium Polymer Battery Systems,” *SAE Technical Paper Series: 2001-01-0959,* 2001.

Ning, G., Haran, B., & Popov, B. N., “Capacity fade study of lithium-ion batteries cycled at high discharge rates,” *Journal of Power Sources*, *117*(1), 160-169, 2003.

Padhi, A.K., Nanjundaswamy, K.S., Goodenough, J.B., "LiFePO4: A Novel Cathode Material for Rechargeable Batteries", *Electrochemical Society Meeting Abstracts*, 96-1, May, 1996.

Pearson, C., Thwaite, C., Curzon, D., Rao, G., “The long-term performance of small-cell batteries without cell balancing electronics”, *Proceedings of the 2004 NASA Battery Workshop,* 2004.

Plett, G. L., “Extended Kalman Filtering for Battery Management Systems of LiPB-based HEV Battery Packs, Part 1: Background,” *Journal of Power Sources*, Vol. 134, pp. 252–261, 2004.

Plett, G. L., “Extended Kalman Filtering for Battery Management Systems of LiPB-based HEV Battery Packs, Part 2: Modeling and Identification,” *Journal of Power Sources*, Vol. 134, pp. 262–276, 2004.

Plett, G. L., “Extended Kalman Filtering for Battery Management Systems of LiPB-based HEV Battery Packs, Part 3: State and Parameter Estimation,” *Journal of Power Sources*, Vol. 134, pp. 277–292, 2004.

Simpson, C., “LM2576, LM3420, LP2951, LP2952: Battery Charging”, *Texas Instruments: Literature Number: SNVA557,* 2011.

Taschner, M. J., “The Development of an Electric Tricycle and Buck-Topology-Based Battery Pack Charger,” *A Thesis Presented to The Graduate Faculty of The University of Akron,* July 2011.

Texas Instruments, “Battery Management Solutions Guide,” *Texas Instruments: SLYT420* 2012.

Vincent, C. A., “Lithium batteries: a 50-year perspective, 1959–2009,” *Solid State Ionics*, *134* (1), 159-167, 2000.

Yevgen, B., “Battery Cell Balancing: What to Balance and How”, Texas Instruments, Last accessed March 2014. <http://focus.ti.com>, 2009.

APPENDICES

APPENDIX A

MATLAB CODE

The following are MATLAB Simulations for various battery pack configurations.

A.1 3-Cells Bypasses Vs 3-Cells w/o Bypasses

```
%% Charge/Discharge with No Bypass

clc;
clear all;
close all;

initialCharge = 65;         % Initial Charge in AmpHours
Ro = -1.09287007554604e-05;
C = 1222821.68974733;
a1 = 15;
a2 = -10.14;
Cd = 25052.2307692307;
Rd = 0.00243491290503841;
tau = 61;

ConstV = 0;
current=0;
chargingTimeHr=4000; %hours
chargingTimeSec = chargingTimeHr*3600;
dischargeCnt=0;
turnOnLoad = 0;

Tsamp = 1; %one sec
x=[ 3.14011797411029       3.14011797411029*0.99
3.14011797411029*1.01;
    3.82610318626407e-07   3.82610318626407e-07*1.01
3.82610318626407e-07*0.99;
    1.53543639979267       1.53543639979267*0.99
1.53543639979267*0.99;
   -1.08581425643197      -1.08581425643197*1.01        -
1.08581425643197*1.01;
    3.76948681201294e-08   3.76948681201294e-08*0.99
3.76948681201294e-08*1.01];
```

```
%I = [zeros(1,1800) -ones(1,3600*3)*20 zeros(1,1800)
ones(1,3600*3)*20 zeros(1,1800)];

qs = zeros(chargingTimeSec,3);
qd = zeros(chargingTimeSec,3);
v = zeros(chargingTimeSec,3);
I = zeros(chargingTimeSec,1);

qs(1,1) = initialCharge*3600*1.02; %Coloumb * Amps*Sec
qd(1,1) = 0;
qs(1,2) = initialCharge*3600; %Coloumb * Amps*Sec
qd(1,2) = 0;
qs(1,3) = initialCharge*3600*0.98; %Coloumb * Amps*Sec
qd(1,3) = 0;

I(1)=0;
v(1,1) =  x(1,1) + x(2,1)*qs(1,1) + x(3,1)*qd(1,1) +
x(4,1)*exp(a2*qs(1,1)/240000)  + x(5,1)*exp(a1*qs(1,1)/240000) +
I(1)*Ro;
v(1,2) =  x(1,2) + x(2,2)*qs(1,2) + x(3,2)*qd(1,2) +
x(4,2)*exp(a2*qs(1,2)/240000)  + x(5,2)*exp(a1*qs(1,2)/240000) +
I(1)*Ro;
v(1,3) =  x(1,3) + x(2,3)*qs(1,3) + x(3,3)*qd(1,3) +
x(4,3)*exp(a2*qs(1,3)/240000)  + x(5,3)*exp(a1*qs(1,3)/240000) +
I(1)*Ro;



for n = 1:1:chargingTimeSec
    for k = 1:3
        if (((v(n,1)+v(n,2)+v(n,3)) < 3.8*3) && ConstV==0)
            current = 20; %%Const. Current
        else
            ConstV = 1;
        end

        if ConstV==1
           vtot = v(n,1)+v(n,2)+v(n,3);
           err = 3.8*3-vtot;
           current = current + 2*err;
           if (current <= 6)
               current = 0;
               turnOnLoad=1;
               ConstV=0;
           end
        end

        if turnOnLoad==1
            I(n) = -60;
            if v(n,1)+v(n,2)+v(n,3)<2.5*3
                turnOnLoad = 0;
                dischargeCnt=dischargeCnt+1;
            end
        else
```

```
            I(n) = current;
        end


        qs(n+1,k) = qs(n,k)+I(n)*Tsamp;
        qd(n+1,k) = qd(n,k)+((I(n)/Cd)-(qd(n,k)/(Rd*Cd)))*Tsamp;
        v(n+1,k) = x(1,k) + x(2,k)*qs(n,k) + x(3,k)*qd(n,k) +
x(4,k)*exp(a2*qs(n,k)/240000)  + x(5,k)*exp(a1*qs(n,k)/240000) +
I(n)*Ro;
    end
end

figure(3)
hold all
plot([0:1:chargingTimeSec]/3600,v(:,1),'r')
plot([0:1:chargingTimeSec]/3600,v(:,2),'g')
plot([0:1:chargingTimeSec]/3600,v(:,3),'c')

xlabel('Time [Hours]')
ylabel('Voltage')
legend({'Cell 1' 'Cell 2' 'Cell 3'})


figure(4)
plot([0:1:chargingTimeSec-1]/3600, I)

xlabel('Time [Hours]')
ylabel('Current')

figure(5)
plot([0:1:chargingTimeSec]/3600, v(:,1)+v(:,2)+v(:,3))
xlabel('Time [Hours]')
ylabel('Voltage')

disp('Discharge cnt is: ')
disp(dischargeCnt)

%% Charge/Discharge with Bypass
clc;
clear all;
close all;


initialCharge = 65;         % Initial Charge in AmpHours
Ro = -1.09287007554604e-05;
C = 1222821.68974733;
a1 = 15;
a2 = -10.14;
Cd = 25052.2307692307;
Rd = 0.00243491290503841;
tau = 61;

ConstV = 0;
current=0;
```

```
chargingTimeHr = 4000; %hours
chargingTimeSec = chargingTimeHr*3600;
dischargeCnt=0;
turnOnLoad = 0;

Tsamp = 1; %one sec
x=[ 3.14011797411029       3.14011797411029*0.99
3.14011797411029*1.01;
    3.82610318626407e-07   3.82610318626407e-07*1.01
3.82610318626407e-07*0.99;
    1.53543639979267       1.53543639979267*0.99
1.53543639979267*0.99;
   -1.08581425643197      -1.08581425643197*1.01        -
1.08581425643197*1.01;
    3.76948681201294e-08   3.76948681201294e-08*0.99
3.76948681201294e-08*1.01];


%I = [zeros(1,1800) -ones(1,3600*3)*20 zeros(1,1800)
ones(1,3600*3)*20 zeros(1,1800)];

qs = zeros(chargingTimeSec,3);
qd = zeros(chargingTimeSec,3);
v = zeros(chargingTimeSec,3);
I = zeros(chargingTimeSec,3);

qs(1,1) = initialCharge*3600*1.02; %Coloumb * Amps*Sec
qd(1,1) = 0;
qs(1,2) = initialCharge*3600; %Coloumb * Amps*Sec
qd(1,2) = 0;
qs(1,3) = initialCharge*3600*0.98; %Coloumb * Amps*Sec
qd(1,3) = 0;


I(1,1)=0;
I(1,2)=0;
I(1,3)=0;

v(1,1) =  x(1,1) + x(2,1)*qs(1,1) + x(3,1)*qd(1,1) +
x(4,1)*exp(a2*qs(1,1)/240000)  + x(5,1)*exp(a1*qs(1,1)/240000) +
I(1)*Ro;
v(1,2) =  x(1,2) + x(2,2)*qs(1,2) + x(3,2)*qd(1,2) +
x(4,2)*exp(a2*qs(1,2)/240000)  + x(5,2)*exp(a1*qs(1,2)/240000) +
I(1)*Ro;
v(1,3) =  x(1,3) + x(2,3)*qs(1,3) + x(3,3)*qd(1,3) +
x(4,3)*exp(a2*qs(1,3)/240000)  + x(5,3)*exp(a1*qs(1,3)/240000) +
I(1)*Ro;


% SIMULATION OF CONTINUOUS CHARGE AND DISCHARGE PROFILES
% SYSTEM STARTS WITH CHARGING
% CHARGE PROFILE IS CONST. CURRENT AT 20 AMPS
% ONCE THE BATTERY PACK TOTAL VOLTAGE (SUM OF EACH INDVIDUAL CELL
VOLTAGE)
```

```
% REACHES 3.8V X 3, THE CHARGE PROFILE GOES TO CONSTANT VOLTAGE
MODE

for n = 1:1:chargingTimeSec

    for k = 1:3   %REPEAT SIMULATION CALCUATIONS FOR EACH CELL
(CELLS IN SERIES ASSUMES, SAME CURRENT THROUGH EACH CELL)

        % CHARGING PROCESS: CONSTANT CURRENT
        % THE FOLLOWING LOGIC DETERMINES WHEN TO SWITCH TO CONSTANT
VOLTAGE MODE
        if (((v(n,1)+v(n,2)+v(n,3)) < 3.8*3) && ConstV==0)
            current = 20; % MAINTAIN CONSTANT CURRENT
        else
            ConstV = 1;   % START CONSTANT VOLTAGE MODE
        end

        % CHARGING PROCESS: CONSTANT VOLTAGE
        % CHECKS IF CURRENT DROPS BELOW 6 AMPS IN CONSTANT VOLTAGE
MODE
        % IF SO, SWITCHES ON LOAD
        if ConstV==1
           vtot = v(n,1)+v(n,2)+v(n,3);
           err = 3.8*3-vtot;
           current = current + 2*err;
           if (current <= 6)
               current = 0;
               turnOnLoad=1;
               ConstV=0;
           end
        end

        % LOAD ACTIVE: 60 AMP LOAD
        % IF LOAD IS ACTIVE, 60 AMP LOAD IS ON AND CHARGING PROCESS
STOPS
        if turnOnLoad==1
            I(n,k) = -60;
            if v(n,1)+v(n,2)+v(n,3)<2.5*3
                turnOnLoad = 0;
                dischargeCnt=dischargeCnt+1;
            end
        else % LOAD IS NOT ON AND CHARGING IS ACTIVE, TURN ON
BYPASS WHEN NECESSARY
            % IMPLEMENTING PASSIVE BYPASS (ASSUMED 1-AMP CONSTANT
CURRENT BYPASS)
            % ACTIVE WHEN CELL VOLTAGE IS GREATER THAN 3.8 VOLTS
            if v(n,k) > 3.8
                I(n,k) = current-1;
            else
                I(n,k) = current;
            end
        end

        % CALCULATE NEXT TIME-STEP USING CELL MODELS
```

```
        qs(n+1,k) = qs(n,k)+I(n,k)*Tsamp;
        qd(n+1,k) = qd(n,k)+((I(n,k)/Cd)-(qd(n,k)/(Rd*Cd)))*Tsamp;
        v(n+1,k) = x(1,k) + x(2,k)*qs(n,k) + x(3,k)*qd(n,k) +
x(4,k)*exp(a2*qs(n,k)/240000)  + x(5,k)*exp(a1*qs(n,k)/240000) +
I(n,k)*Ro;
    end
end

figure(5)
hold all
plot([0:1:chargingTimeSec]/3600,v(:,1),'r')
plot([0:1:chargingTimeSec]/3600,v(:,2),'g')
plot([0:1:chargingTimeSec]/3600,v(:,3),'c')

xlabel('Time [Hours]')
ylabel('Voltage')

figure(6)
plot([0:1:chargingTimeSec-1]/3600, I(:,1))

xlabel('Time [Hours]')
ylabel('Current')

disp('Discharge cnt is: ')
disp(dischargeCnt)
```

A.2 MATLAB Simulations: 10-Cells & Integrated Charger & Battery Pack w/ Bypasses

```
%% Charge/Discharge with Bypass 10 Cells Integrated charger
clc;
clear all;
%close all;

% VALUES DERIVED FROM ACTUAL CHARGE DATA, INTEGRATED
CHARGING CURRENT
% FROM EACH CELL (TOP TO BOTTOM IS CELL 1 TO 10)

% ampHour =
%         53.4478888888888
%         52.4618888888889
%         51.3797222222222
%         51.1152222222222
%         50.5888333333333
%         51.4822499999999
%         51.1220555555555
%         50.7726944444444
%         49.8306944444444
%         50.1831111111111

% SOC_Percent =
%         89.0798148148147
%         87.4364814814814
%         85.6328703703704
%         85.192037037037
%         84.3147222222221
%         85.8037499999999
%         85.2034259259258
%         84.6211574074073
%         83.0511574074073
%         83.6385185185185

% Initial_Charge_SOC_Percent =
%         10.9201851851853
%         12.5635185185186
%         14.3671296296296
%          14.807962962963
%         15.6852777777779
```

```
%           14.1962500000001
%           14.7965740740742
%           15.3788425925927
%           16.9488425925927
%           16.3614814814815

initialCharge = 60;       % Initial Charge in AmpHours
Ro = -1.09287007554604e-05;
C = 1222821.68974733;
a1 = 15;
a2 = -10.14;
Cd = 25052.2307692307;
Rd = 0.00243491290503841;
tau = Cd*Rd;

ConstV = 0;
current=0;
chargingTimeHr = 50; %Simulation hours
chargingTimeSec = chargingTimeHr*3600;
dischargeCnt=0;
turnOnLoad = 0;

Tsamp = 1; %one sec

x=[ 3.14011797411029;
    3.82610318626407e-07;
    1.53543639979267;
   -1.08581425643197;
    3.76948681201294e-08];

x = [x x x x x x x x x];


%I = [zeros(1,1800) -ones(1,3600*3)*20 zeros(1,1800) ones(1,3600*3)*20
zeros(1,1800)];

qs = zeros(chargingTimeSec,10);
qd = zeros(chargingTimeSec,10);
v = zeros(chargingTimeSec,10);
I = zeros(chargingTimeSec,10);


%           10.9201851851853
%           12.5635185185186
%           14.3671296296296
%            14.807962962963
```

```
%       15.6852777777779
%       14.1962500000001
%       14.7965740740742
%       15.3788425925927
%       16.9488425925927
%       16.3614814814815

qs(1,1) = initialCharge*3600*0.08; %Coloumb * Amps*Sec
qd(1,1) = 0;
qs(1,2) = initialCharge*3600*0.126; %Coloumb * Amps*Sec
qd(1,2) = 0;
qs(1,3) = initialCharge*3600*0.144; %Coloumb * Amps*Sec
qd(1,3) = 0;
qs(1,4) = initialCharge*3600*0.148; %Coloumb * Amps*Sec
qd(1,4) = 0;
qs(1,5) = initialCharge*3600*0.157; %Coloumb * Amps*Sec
qd(1,5) = 0;
qs(1,6) = initialCharge*3600*0.142; %Coloumb * Amps*Sec
qd(1,6) = 0;
qs(1,7) = initialCharge*3600*0.148; %Coloumb * Amps*Sec
qd(1,7) = 0;
qs(1,8) = initialCharge*3600*0.154; %Coloumb * Amps*Sec
qd(1,8) = 0;
qs(1,9) = initialCharge*3600*0.169; %Coloumb * Amps*Sec
qd(1,9) = 0;
qs(1,10) = initialCharge*3600*0.164; %Coloumb * Amps*Sec
qd(1,10) = 0;

I(1,1)=0;
I(1,2)=0;
I(1,3)=0;
I(1,4)=0;
I(1,5)=0;
I(1,6)=0;
I(1,7)=0;
I(1,8)=0;
I(1,9)=0;
I(1,10)=0;


v(1,1)  =  x(1,1)  + x(2,1) *qs(1,1)  + x(3,1)*qd(1,1)   + x(4,1)
*exp(a2*qs(1,1)/240000)  + x(5,1) *exp(a1*qs(1,1)/240000)  + I(1)*Ro;
v(1,2)  =  x(1,2)  + x(2,2) *qs(1,2)  + x(3,2)*qd(1,2)   + x(4,2)
*exp(a2*qs(1,2)/240000)  + x(5,2) *exp(a1*qs(1,2)/240000)  + I(1)*Ro;
v(1,3)  =  x(1,3)  + x(2,3) *qs(1,3)  + x(3,3)*qd(1,3)   + x(4,3)
*exp(a2*qs(1,3)/240000)  + x(5,3) *exp(a1*qs(1,3)/240000)  + I(1)*Ro;
```

```
v(1,4)  =  x(1,4)  + x(2,4) *qs(1,4)  + x(3,4)*qd(1,4)   + x(4,4)
*exp(a2*qs(1,4)/240000)  + x(5,4) *exp(a1*qs(1,4)/240000)  + I(1)*Ro;
v(1,5)  =  x(1,5)  + x(2,5) *qs(1,5)  + x(3,5)*qd(1,5)   + x(4,5)
*exp(a2*qs(1,5)/240000)  + x(5,5) *exp(a1*qs(1,5)/240000)  + I(1)*Ro;
v(1,6)  =  x(1,6)  + x(2,6) *qs(1,6)  + x(3,6)*qd(1,6)   + x(4,6)
*exp(a2*qs(1,6)/240000)  + x(5,6) *exp(a1*qs(1,6)/240000)  + I(1)*Ro;
v(1,7)  =  x(1,7)  + x(2,7) *qs(1,7)  + x(3,7)*qd(1,7)   + x(4,7)
*exp(a2*qs(1,7)/240000)  + x(5,7) *exp(a1*qs(1,7)/240000)  + I(1)*Ro;
v(1,8)  =  x(1,8)  + x(2,8) *qs(1,8)  + x(3,8)*qd(1,8)   + x(4,8)
*exp(a2*qs(1,8)/240000)  + x(5,8) *exp(a1*qs(1,8)/240000)  + I(1)*Ro;
v(1,9)  =  x(1,9)  + x(2,9) *qs(1,9)  + x(3,9)*qd(1,9)   + x(4,9)
*exp(a2*qs(1,9)/240000)  + x(5,9) *exp(a1*qs(1,9)/240000)  + I(1)*Ro;
v(1,10) =  x(1,10) + x(2,10)*qs(1,10) + x(3,10)*qd(1,10) +
x(4,10)*exp(a2*qs(1,10)/240000) + x(5,10)*exp(a1*qs(1,10)/240000) + I(1)*Ro;


hold_balancing_current = 0;

% SIMULATION OF CONTINUOUS CHARGE AND DISCHARGE PROFILES
% SYSTEM STARTS WITH CHARGING
% CHARGE PROFILE IS CONST. CURRENT AT 20 AMPS
% ONCE THE BATTERY PACK TOTAL VOLTAGE (SUM OF EACH
INDVIDUAL CELL VOLTAGE)
% REACHES 3.8V X 3, THE CHARGE PROFILE GOES TO CONSTANT
VOLTAGE MODE

bypass_current = zeros(chargingTimeSec,10);
previous_bypass_current = zeros(10,1);
bypass_begun = zeros(10,1);

for n = 1:1:chargingTimeSec

   for k = 1:10   %REPEAT SIMULATION CALCUATIONS FOR EACH CELL
(CELLS IN SERIES ASSUMES, SAME CURRENT THROUGH EACH CELL)

      % CHARGING PROCESS: CONSTANT CURRENT
      % THE FOLLOWING LOGIC DETERMINES WHEN TO SWITCH TO
CONSTANT VOLTAGE MODE
      if (max(v(n,:)) <= 3.8 && ConstV==0)
         charge_current = 12; % MAINTAIN CONSTANT CURRENT
      else
         ConstV = 1;   % START CONSTANT VOLTAGE MODE
      end

      % CHARGING PROCESS: CONSTANT VOLTAGE
```

```
        % CHECKS IF CURRENT DROPS BELOW 6 AMPS IN CONSTANT
VOLTAGE MODE
        % IF SO, SWITCHES ON LOAD
        if ConstV==1
          %vtot =
v(n,1)+v(n,2)+v(n,3)+v(n,4)+v(n,5)+v(n,6)+v(n,7)+v(n,8)+v(n,9)+v(n,10);
          if max(v(n,:)) >= 4.2
             if charge_current > 2 && (hold_balancing_current ~= 1)
                 charge_current = charge_current-2;
             else
                 charge_current = 2.0;
                 hold_balancing_current = 1;
             end
          end

          if (min(v(n,:)) >= 3.79)
             current = 0;
             turnOnLoad=1;
             ConstV=0;
          end
        end

        % LOAD ACTIVE: 60 AMP LOAD
        % IF LOAD IS ACTIVE, 60 AMP LOAD IS ON AND CHARGING
PROCESS STOPS
        if turnOnLoad==1
           I(n,k) = -60;
           if
(v(n,1)+v(n,2)+v(n,3)+v(n,4)+v(n,5)+v(n,6)+v(n,7)+v(n,8)+v(n,9)+v(n,10)) <
(2.5*10)
              turnOnLoad = 0;
              dischargeCnt=dischargeCnt+1;
              charge_current = 12.0;
              hold_balancing_current=0;
              bypass_begun = zeros(10,1);
           end
        else % LOAD IS NOT ON AND CHARGING IS ACTIVE, TURN ON
BYPASS WHEN NECESSARY
           % IMPLEMENTING PASSIVE BYPASS (ASSUMED 1-AMP
CONSTANT CURRENT BYPASS)
           % ACTIVE WHEN CELL VOLTAGE IS GREATER THAN 3.8 VOLTS
           if v(n,k) > 3.8 && (bypass_begun(k) ~= 1)
              bypass_begun(k) = 1;
              error = 3.8-v(n,k);
              kp = 100;
              bypass_current(n,k) = bypass_current(n-1,k) - error*kp;
```

```
            if bypass_current(n,k) > 2
                bypass_current(n,k) = 2;
            end

            I(n,k) = charge_current - bypass_current(n,k);

        elseif bypass_begun(k)==1
            %I(n,k) = charge_current-2;
            bypass_current(n,k) = 2;
            I(n,k) = charge_current - bypass_current(n,k);
        else
            I(n,k) = charge_current;
        end
    end

    if v(n,:) >= 3.8
        break
    end

    % CALCULATE NEXT TIME-STEP USING CELL MODELS
    qs(n+1,k) = qs(n,k)+I(n,k)*Tsamp;
    qd(n+1,k) = qd(n,k)+((I(n,k)/Cd)-(qd(n,k)/(Rd*Cd)))*Tsamp;
    v(n+1,k) = x(1,k) + x(2,k)*qs(n,k) + x(3,k)*qd(n,k) +
x(4,k)*exp(a2*qs(n,k)/240000) + x(5,k)*exp(a1*qs(n,k)/240000) + I(n,k)*Ro;
  end
end

figure(4)
subplot(3,1,1)
hold all
plot([0:1:chargingTimeSec]/3600,v(:,1))
plot([0:1:chargingTimeSec]/3600,v(:,2))
plot([0:1:chargingTimeSec]/3600,v(:,3))
plot([0:1:chargingTimeSec]/3600,v(:,4))
plot([0:1:chargingTimeSec]/3600,v(:,5))
plot([0:1:chargingTimeSec]/3600,v(:,6))
plot([0:1:chargingTimeSec]/3600,v(:,7))
plot([0:1:chargingTimeSec]/3600,v(:,8))
plot([0:1:chargingTimeSec]/3600,v(:,9))
plot([0:1:chargingTimeSec]/3600,v(:,10))
plot([0:1:chargingTimeSec-1]/3600,4.2*ones(chargingTimeSec,1),'r')
plot([0:1:chargingTimeSec-1]/3600,2.5*ones(chargingTimeSec,1),'r')
% legend('Cell1','Cell2','Cell3','Cell4','Cell5','Cell6','Cell7','Cell8','Cell9','Cell10')
xlabel('Time [Hours]')
ylabel('Voltage')
```

```
subplot(3,1,2)
hold all
plot([0:1:chargingTimeSec-1]/3600, I(:,1))
plot([0:1:chargingTimeSec-1]/3600, I(:,2))
plot([0:1:chargingTimeSec-1]/3600, I(:,3))
plot([0:1:chargingTimeSec-1]/3600, I(:,4))
plot([0:1:chargingTimeSec-1]/3600, I(:,5))
plot([0:1:chargingTimeSec-1]/3600, I(:,6))
plot([0:1:chargingTimeSec-1]/3600, I(:,7))
plot([0:1:chargingTimeSec-1]/3600, I(:,8))
plot([0:1:chargingTimeSec-1]/3600, I(:,9))
plot([0:1:chargingTimeSec-1]/3600, I(:,10))

xlabel('Time [Hours]')
ylabel('Current')
disp('Discharge cnt is: ')
disp(dischargeCnt)

subplot(3,1,3)
hold all
plot([0:1:chargingTimeSec-1]/3600, bypass_current(:,1))
plot([0:1:chargingTimeSec-1]/3600, bypass_current(:,2))
plot([0:1:chargingTimeSec-1]/3600, bypass_current(:,3))
plot([0:1:chargingTimeSec-1]/3600, bypass_current(:,4))
plot([0:1:chargingTimeSec-1]/3600, bypass_current(:,5))
plot([0:1:chargingTimeSec-1]/3600, bypass_current(:,6))
plot([0:1:chargingTimeSec-1]/3600, bypass_current(:,7))
plot([0:1:chargingTimeSec-1]/3600, bypass_current(:,8))
plot([0:1:chargingTimeSec-1]/3600, bypass_current(:,9))
plot([0:1:chargingTimeSec-1]/3600, bypass_current(:,10))

xlabel('Time [Hours]')
ylabel('Bypass Current')
```

A.3 MATLAB Simulations: 10-Cells, Simple Charger, and with Bypasses

```
%% Charge/Discharge with Bypass 10 Cells
clc;
clear all;
%close all;

% VALUES DERIVED FROM ACTUAL CHARGE DATA, INTEGRATED
CHARGING CURRENT
% FROM EACH CELL (TOP TO BOTTOM IS CELL 1 TO 10)

% ampHour =
%         53.4478888888888
%         52.4618888888889
%         51.3797222222222
%         51.1152222222222
%         50.5888333333333
%         51.4822499999999
%         51.1220555555555
%         50.7726944444444
%         49.8306944444444
%         50.1831111111111

% SOC_Percent =
%         89.0798148148147
%         87.4364814814814
%         85.6328703703704
%         85.192037037037
%         84.3147222222221
%         85.8037499999999
%         85.2034259259258
%         84.6211574074073
%         83.0511574074073
%         83.6385185185185

% Initial_Charge_SOC_Percent =
%         10.9201851851853
%         12.5635185185186
%         14.3671296296296
%          14.807962962963
%         15.6852777777779
%         14.1962500000001
%         14.7965740740742
%         15.3788425925927
```

```
%           16.9488425925927
%           16.3614814814815

initialCharge = 60;       % Initial Charge in AmpHours
Ro = -1.09287007554604e-05;
C = 1222821.68974733;
a1 = 15;
a2 = -10.14;
Cd = 25052.2307692307;
Rd = 0.00243491290503841;
%Cd = 7000;
%Rd = 0.005;
tau = Cd*Rd;

ConstV = 0;
current=0;
chargingTimeHr = 120; %Simulation hours
chargingTimeSec = chargingTimeHr*3600;
dischargeCnt=0;
turnOnLoad = 0;

Tsamp = 1; %one sec

x=[ 3.14011797411029;
    3.82610318626407e-07;
    1.53543639979267;
   -1.08581425643197;
    3.76948681201294e-08];

x = [x x x x x x x x x];


%I = [zeros(1,1800) -ones(1,3600*3)*20 zeros(1,1800) ones(1,3600*3)*20
zeros(1,1800)];

qs = zeros(chargingTimeSec,10);
qd = zeros(chargingTimeSec,10);
v = zeros(chargingTimeSec,10);
I = zeros(chargingTimeSec,10);


%           10.9201851851853
%           12.5635185185186
%           14.3671296296296
%            14.807962962963
%           15.6852777777779
```

```
%          14.1962500000001
%          14.7965740740742
%          15.3788425925927
%          16.9488425925927
%          16.3614814814815

qs(1,1) = initialCharge*3600*0.08; %Coloumb * Amps*Sec
qd(1,1) = 0;
qs(1,2) = initialCharge*3600*0.126; %Coloumb * Amps*Sec
qd(1,2) = 0;
qs(1,3) = initialCharge*3600*0.144; %Coloumb * Amps*Sec
qd(1,3) = 0;
qs(1,4) = initialCharge*3600*0.148; %Coloumb * Amps*Sec
qd(1,4) = 0;
qs(1,5) = initialCharge*3600*0.157; %Coloumb * Amps*Sec
qd(1,5) = 0;
qs(1,6) = initialCharge*3600*0.142; %Coloumb * Amps*Sec
qd(1,6) = 0;
qs(1,7) = initialCharge*3600*0.148; %Coloumb * Amps*Sec
qd(1,7) = 0;
qs(1,8) = initialCharge*3600*0.154; %Coloumb * Amps*Sec
qd(1,8) = 0;
qs(1,9) = initialCharge*3600*0.169; %Coloumb * Amps*Sec
qd(1,9) = 0;
qs(1,10) = initialCharge*3600*0.164; %Coloumb * Amps*Sec
qd(1,10) = 0;

I(1,1)=0;
I(1,2)=0;
I(1,3)=0;
I(1,4)=0;
I(1,5)=0;
I(1,6)=0;
I(1,7)=0;
I(1,8)=0;
I(1,9)=0;
I(1,10)=0;


v(1,1)  =  x(1,1)  + x(2,1) *qs(1,1)  + x(3,1)*qd(1,1)   + x(4,1)
*exp(a2*qs(1,1)/240000)  + x(5,1) *exp(a1*qs(1,1)/240000)  + I(1)*Ro;
v(1,2)  =  x(1,2)  + x(2,2) *qs(1,2)  + x(3,2)*qd(1,2)   + x(4,2)
*exp(a2*qs(1,2)/240000)  + x(5,2) *exp(a1*qs(1,2)/240000)  + I(1)*Ro;
v(1,3)  =  x(1,3)  + x(2,3) *qs(1,3)  + x(3,3)*qd(1,3)   + x(4,3)
*exp(a2*qs(1,3)/240000)  + x(5,3) *exp(a1*qs(1,3)/240000)  + I(1)*Ro;
```

```
v(1,4)  =  x(1,4)  + x(2,4) *qs(1,4)  + x(3,4)*qd(1,4)   + x(4,4)
*exp(a2*qs(1,4)/240000)  + x(5,4) *exp(a1*qs(1,4)/240000)  + I(1)*Ro;
v(1,5)  =  x(1,5)  + x(2,5) *qs(1,5)  + x(3,5)*qd(1,5)   + x(4,5)
*exp(a2*qs(1,5)/240000)  + x(5,5) *exp(a1*qs(1,5)/240000)  + I(1)*Ro;
v(1,6)  =  x(1,6)  + x(2,6) *qs(1,6)  + x(3,6)*qd(1,6)   + x(4,6)
*exp(a2*qs(1,6)/240000)  + x(5,6) *exp(a1*qs(1,6)/240000)  + I(1)*Ro;
v(1,7)  =  x(1,7)  + x(2,7) *qs(1,7)  + x(3,7)*qd(1,7)   + x(4,7)
*exp(a2*qs(1,7)/240000)  + x(5,7) *exp(a1*qs(1,7)/240000)  + I(1)*Ro;
v(1,8)  =  x(1,8)  + x(2,8) *qs(1,8)  + x(3,8)*qd(1,8)   + x(4,8)
*exp(a2*qs(1,8)/240000)  + x(5,8) *exp(a1*qs(1,8)/240000)  + I(1)*Ro;
v(1,9)  =  x(1,9)  + x(2,9) *qs(1,9)  + x(3,9)*qd(1,9)   + x(4,9)
*exp(a2*qs(1,9)/240000)  + x(5,9) *exp(a1*qs(1,9)/240000)  + I(1)*Ro;
v(1,10) =  x(1,10) + x(2,10)*qs(1,10) + x(3,10)*qd(1,10) +
x(4,10)*exp(a2*qs(1,10)/240000) + x(5,10)*exp(a1*qs(1,10)/240000) + I(1)*Ro;


hold_balancing_current = 0;

% SIMULATION OF CONTINUOUS CHARGE AND DISCHARGE PROFILES
% SYSTEM STARTS WITH CHARGING
% CHARGE PROFILE IS CONST. CURRENT AT 20 AMPS
% ONCE THE BATTERY PACK TOTAL VOLTAGE (SUM OF EACH
INDVIDUAL CELL VOLTAGE)
% REACHES 3.8V X 3, THE CHARGE PROFILE GOES TO CONSTANT
VOLTAGE MODE

bypass_current = zeros(chargingTimeSec,10);
previous_bypass_current = zeros(10,1);
bypass_begun = zeros(10,1);

for n = 1:1:chargingTimeSec

   for k = 1:10   %REPEAT SIMULATION CALCUATIONS FOR EACH CELL
(CELLS IN SERIES ASSUMES, SAME CURRENT THROUGH EACH CELL)

      % CHARGING PROCESS: CONSTANT CURRENT
      % THE FOLLOWING LOGIC DETERMINES WHEN TO SWITCH TO
CONSTANT VOLTAGE MODE
      if (max(v(n,:)) <= 3.8 && ConstV==0)
         charge_current = 12; % MAINTAIN CONSTANT CURRENT
      else
         ConstV = 1;   % START CONSTANT VOLTAGE MODE
      end

      % CHARGING PROCESS: CONSTANT VOLTAGE
```

```
      % CHECKS IF CURRENT DROPS BELOW 6 AMPS IN CONSTANT
VOLTAGE MODE
      % IF SO, SWITCHES ON LOAD
      if ConstV==1
        vtot =
v(n,1)+v(n,2)+v(n,3)+v(n,4)+v(n,5)+v(n,6)+v(n,7)+v(n,8)+v(n,9)+v(n,10);

        if vtot >= 4.2*10
           current = 0;
           turnOnLoad=1;
           ConstV=0;
        end
      end

      % LOAD ACTIVE: 60 AMP LOAD
      % IF LOAD IS ACTIVE, 60 AMP LOAD IS ON AND CHARGING
PROCESS STOPS
      if turnOnLoad==1
         I(n,k) = -60;
         if
(v(n,1)+v(n,2)+v(n,3)+v(n,4)+v(n,5)+v(n,6)+v(n,7)+v(n,8)+v(n,9)+v(n,10)) <
(2.5*10)
            turnOnLoad = 0;
            dischargeCnt=dischargeCnt+1;
            charge_current = 12.0;
            bypass_begun = zeros(10,1);
         end
      else % LOAD IS NOT ON AND CHARGING IS ACTIVE, TURN ON
BYPASS WHEN NECESSARY
         % IMPLEMENTING PASSIVE BYPASS (ASSUMED 1-AMP
CONSTANT CURRENT BYPASS)
         % ACTIVE WHEN CELL VOLTAGE IS GREATER THAN 3.8 VOLTS
         if v(n,k) > 3.8 && (bypass_begun(k) ~= 1)
            bypass_begun(k) = 1;
            error = 3.8-v(n,k);
            kp = 100;
            bypass_current(n,k) = bypass_current(n-1,k) - error*kp;

            if bypass_current(n,k) > 2
               bypass_current(n,k) = 2;
            end

            I(n,k) = charge_current - bypass_current(n,k);

         elseif bypass_begun(k)==1
            %I(n,k) = charge_current-2;
```

```
            bypass_current(n,k) = 2;
            I(n,k) = charge_current - bypass_current(n,k);
        else
            I(n,k) = charge_current;
        end
    end


    % CALCULATE NEXT TIME-STEP USING CELL MODELS
    qs(n+1,k) = qs(n,k)+I(n,k)*Tsamp;
    qd(n+1,k) = qd(n,k)+((I(n,k)/Cd)-(qd(n,k)/(Rd*Cd)))*Tsamp;
    v(n+1,k) = x(1,k) + x(2,k)*qs(n,k) + x(3,k)*qd(n,k) +
x(4,k)*exp(a2*qs(n,k)/240000) + x(5,k)*exp(a1*qs(n,k)/240000) + I(n,k)*Ro;
  end
end

figure(5)
subplot(3,1,1)
hold all
plot([0:1:chargingTimeSec]/3600,v(:,1))
plot([0:1:chargingTimeSec]/3600,v(:,2))
plot([0:1:chargingTimeSec]/3600,v(:,3))
plot([0:1:chargingTimeSec]/3600,v(:,4))
plot([0:1:chargingTimeSec]/3600,v(:,5))
plot([0:1:chargingTimeSec]/3600,v(:,6))
plot([0:1:chargingTimeSec]/3600,v(:,7))
plot([0:1:chargingTimeSec]/3600,v(:,8))
plot([0:1:chargingTimeSec]/3600,v(:,9))
plot([0:1:chargingTimeSec]/3600,v(:,10))
plot([0:1:chargingTimeSec-1]/3600,4.2*ones(chargingTimeSec,1),'r')
plot([0:1:chargingTimeSec-1]/3600,2.5*ones(chargingTimeSec,1),'r')
% legend('Cell1','Cell2','Cell3','Cell4','Cell5','Cell6','Cell7','Cell8','Cell9','Cell10')
xlabel('Time [Hours]')
ylabel('Voltage')

subplot(3,1,2)
hold all
plot([0:1:chargingTimeSec-1]/3600, I(:,1))
plot([0:1:chargingTimeSec-1]/3600, I(:,2))
plot([0:1:chargingTimeSec-1]/3600, I(:,3))
plot([0:1:chargingTimeSec-1]/3600, I(:,4))
plot([0:1:chargingTimeSec-1]/3600, I(:,5))
plot([0:1:chargingTimeSec-1]/3600, I(:,6))
plot([0:1:chargingTimeSec-1]/3600, I(:,7))
plot([0:1:chargingTimeSec-1]/3600, I(:,8))
plot([0:1:chargingTimeSec-1]/3600, I(:,9))
```

```
plot([0:1:chargingTimeSec-1]/3600, I(:,10))

xlabel('Time [Hours]')
ylabel('Current')
% legend('Cell1','Cell2','Cell3','Cell4','Cell5','Cell6','Cell7','Cell8','Cell9','Cell10')
disp('Discharge cnt is: ')
disp(dischargeCnt)

subplot(3,1,3)
hold all
plot([0:1:chargingTimeSec-1]/3600, bypass_current(:,1))
plot([0:1:chargingTimeSec-1]/3600, bypass_current(:,2))
plot([0:1:chargingTimeSec-1]/3600, bypass_current(:,3))
plot([0:1:chargingTimeSec-1]/3600, bypass_current(:,4))
plot([0:1:chargingTimeSec-1]/3600, bypass_current(:,5))
plot([0:1:chargingTimeSec-1]/3600, bypass_current(:,6))
plot([0:1:chargingTimeSec-1]/3600, bypass_current(:,7))
plot([0:1:chargingTimeSec-1]/3600, bypass_current(:,8))
plot([0:1:chargingTimeSec-1]/3600, bypass_current(:,9))
plot([0:1:chargingTimeSec-1]/3600, bypass_current(:,10))

xlabel('Time [Hours]')
ylabel('Bypass Current')
% legend('Cell1','Cell2','Cell3','Cell4','Cell5','Cell6','Cell7','Cell8','Cell9','Cell10')
```

A.4 MATLAB Simulations: 10-Cells, Simple Charger, and No Bypasses

```
%% Charge/Discharge with No Bypass 10 Cells
clc;
clear all;
%%close all;

% VALUES DERIVED FROM ACTUAL CHARGE DATA, INTEGRATED
CHARGING CURRENT
% FROM EACH CELL (TOP TO BOTTOM IS CELL 1 TO 10)

% ampHour =
%         53.4478888888888
%         52.4618888888889
%         51.3797222222222
%         51.1152222222222
%         50.5888333333333
%         51.4822499999999
%         51.1220555555555
%         50.7726944444444
%         49.8306944444444
%         50.1831111111111

% SOC_Percent =
%         89.0798148148147
%         87.4364814814814
%         85.6328703703704
%         85.192037037037
%         84.3147222222221
%         85.8037499999999
%         85.2034259259258
%         84.6211574074073
%         83.0511574074073
%         83.6385185185185

% Initial_Charge_SOC_Percent =
%         10.9201851851853
%         12.5635185185186
%         14.3671296296296
%          14.807962962963
%         15.6852777777779
%         14.1962500000001
%         14.7965740740742
%         15.3788425925927
```

```
%           16.9488425925927
%           16.3614814814815

initialCharge = 60;       % Initial Charge in AmpHours
Ro = -1.09287007554604e-05;
C = 1222821.68974733;
a1 = 15;
a2 = -10.14;
Cd = 25052.2307692307;
Rd = 0.00243491290503841;
%Cd = 7000;
%Rd = 0.005;
tau = Cd*Rd;

ConstV = 0;
current=0;
chargingTimeHr = 100; %Simulation hours
chargingTimeSec = chargingTimeHr*3600;
dischargeCnt=0;
turnOnLoad = 0;

Tsamp = 1; %one sec

x=[ 3.14011797411029;
    3.82610318626407e-07;
    1.53543639979267;
   -1.08581425643197;
    3.76948681201294e-08];

x = [x x x x x x x x x];


%I = [zeros(1,1800) -ones(1,3600*3)*20 zeros(1,1800) ones(1,3600*3)*20
zeros(1,1800)];

qs = zeros(chargingTimeSec,10);
qd = zeros(chargingTimeSec,10);
v = zeros(chargingTimeSec,10);
I = zeros(chargingTimeSec,10);


%           10.9201851851853
%           12.5635185185186
%           14.3671296296296
%           14.807962962963
%           15.6852777777779
```

```
%          14.1962500000001
%          14.7965740740742
%          15.3788425925927
%          16.9488425925927
%          16.3614814814815

qs(1,1) = initialCharge*3600*0.08; %Coloumb * Amps*Sec
qd(1,1) = 0;
qs(1,2) = initialCharge*3600*0.126; %Coloumb * Amps*Sec
qd(1,2) = 0;
qs(1,3) = initialCharge*3600*0.144; %Coloumb * Amps*Sec
qd(1,3) = 0;
qs(1,4) = initialCharge*3600*0.148; %Coloumb * Amps*Sec
qd(1,4) = 0;
qs(1,5) = initialCharge*3600*0.157; %Coloumb * Amps*Sec
qd(1,5) = 0;
qs(1,6) = initialCharge*3600*0.142; %Coloumb * Amps*Sec
qd(1,6) = 0;
qs(1,7) = initialCharge*3600*0.148; %Coloumb * Amps*Sec
qd(1,7) = 0;
qs(1,8) = initialCharge*3600*0.154; %Coloumb * Amps*Sec
qd(1,8) = 0;
qs(1,9) = initialCharge*3600*0.169; %Coloumb * Amps*Sec
qd(1,9) = 0;
qs(1,10) = initialCharge*3600*0.164; %Coloumb * Amps*Sec
qd(1,10) = 0;

I(1,1)=0;
I(1,2)=0;
I(1,3)=0;
I(1,4)=0;
I(1,5)=0;
I(1,6)=0;
I(1,7)=0;
I(1,8)=0;
I(1,9)=0;
I(1,10)=0;


v(1,1)  =  x(1,1)  + x(2,1) *qs(1,1)  + x(3,1)*qd(1,1)   + x(4,1)
*exp(a2*qs(1,1)/240000)  + x(5,1) *exp(a1*qs(1,1)/240000)  + I(1)*Ro;
v(1,2)  =  x(1,2)  + x(2,2) *qs(1,2)  + x(3,2)*qd(1,2)   + x(4,2)
*exp(a2*qs(1,2)/240000)  + x(5,2) *exp(a1*qs(1,2)/240000)  + I(1)*Ro;
v(1,3)  =  x(1,3)  + x(2,3) *qs(1,3)  + x(3,3)*qd(1,3)   + x(4,3)
*exp(a2*qs(1,3)/240000)  + x(5,3) *exp(a1*qs(1,3)/240000)  + I(1)*Ro;
```

```
v(1,4)  =  x(1,4)  + x(2,4) *qs(1,4)  + x(3,4)*qd(1,4)   + x(4,4)
*exp(a2*qs(1,4)/240000)  + x(5,4) *exp(a1*qs(1,4)/240000)  + I(1)*Ro;
v(1,5)  =  x(1,5)  + x(2,5) *qs(1,5)  + x(3,5)*qd(1,5)   + x(4,5)
*exp(a2*qs(1,5)/240000)  + x(5,5) *exp(a1*qs(1,5)/240000)  + I(1)*Ro;
v(1,6)  =  x(1,6)  + x(2,6) *qs(1,6)  + x(3,6)*qd(1,6)   + x(4,6)
*exp(a2*qs(1,6)/240000)  + x(5,6) *exp(a1*qs(1,6)/240000)  + I(1)*Ro;
v(1,7)  =  x(1,7)  + x(2,7) *qs(1,7)  + x(3,7)*qd(1,7)   + x(4,7)
*exp(a2*qs(1,7)/240000)  + x(5,7) *exp(a1*qs(1,7)/240000)  + I(1)*Ro;
v(1,8)  =  x(1,8)  + x(2,8) *qs(1,8)  + x(3,8)*qd(1,8)   + x(4,8)
*exp(a2*qs(1,8)/240000)  + x(5,8) *exp(a1*qs(1,8)/240000)  + I(1)*Ro;
v(1,9)  =  x(1,9)  + x(2,9) *qs(1,9)  + x(3,9)*qd(1,9)   + x(4,9)
*exp(a2*qs(1,9)/240000)  + x(5,9) *exp(a1*qs(1,9)/240000)  + I(1)*Ro;
v(1,10) =  x(1,10) + x(2,10)*qs(1,10) + x(3,10)*qd(1,10) +
x(4,10)*exp(a2*qs(1,10)/240000) + x(5,10)*exp(a1*qs(1,10)/240000) + I(1)*Ro;


% SIMULATION OF CONTINUOUS CHARGE AND DISCHARGE PROFILES
% SYSTEM STARTS WITH CHARGING
% CHARGE PROFILE IS CONST. CURRENT AT 20 AMPS
% ONCE THE BATTERY PACK TOTAL VOLTAGE (SUM OF EACH
INDVIDUAL CELL VOLTAGE)
% REACHES 3.8V X 3, THE CHARGE PROFILE GOES TO CONSTANT
VOLTAGE MODE

for n = 1:1:chargingTimeSec

   for k = 1:10   %REPEAT SIMULATION CALCUATIONS FOR EACH CELL
(CELLS IN SERIES ASSUMES, SAME CURRENT THROUGH EACH CELL)

      % CHARGING PROCESS: CONSTANT CURRENT
      % THE FOLLOWING LOGIC DETERMINES WHEN TO SWITCH TO
CONSTANT VOLTAGE MODE
      if (max(v(n,:)) <= 3.8 && ConstV==0)
         charge_current = 12; % MAINTAIN CONSTANT CURRENT
      else
         ConstV = 1;   % START CONSTANT VOLTAGE MODE
      end

      % CHARGING PROCESS: CONSTANT VOLTAGE
      % CHECKS IF CURRENT DROPS BELOW 6 AMPS IN CONSTANT
VOLTAGE MODE
      % IF SO, SWITCHES ON LOAD
      if ConstV==1
         vtot =
v(n,1)+v(n,2)+v(n,3)+v(n,4)+v(n,5)+v(n,6)+v(n,7)+v(n,8)+v(n,9)+v(n,10);
         if vtot >= 4.2*10
```

```
            current = 0;
            turnOnLoad=1;
            ConstV=0;
          end
       end

       % LOAD ACTIVE: 60 AMP LOAD
       % IF LOAD IS ACTIVE, 60 AMP LOAD IS ON AND CHARGING
PROCESS STOPS
       if turnOnLoad==1
          I(n,k) = -60;
          if
(v(n,1)+v(n,2)+v(n,3)+v(n,4)+v(n,5)+v(n,6)+v(n,7)+v(n,8)+v(n,9)+v(n,10)) <
(2.5*10)
             turnOnLoad = 0;
             dischargeCnt=dischargeCnt+1
             charge_current = 12;
          end
       else % LOAD IS NOT ON AND CHARGING IS ACTIVE, TURN ON
BYPASS WHEN NECESSARY

          I(n,k) = charge_current;
       end


       % CALCULATE NEXT TIME-STEP USING CELL MODELS
       qs(n+1,k) = qs(n,k)+I(n,k)*Tsamp;
       qd(n+1,k) = qd(n,k)+((I(n,k)/Cd)-(qd(n,k)/(Rd*Cd)))*Tsamp;
       v(n+1,k) = x(1,k) + x(2,k)*qs(n,k) + x(3,k)*qd(n,k) +
x(4,k)*exp(a2*qs(n,k)/240000) + x(5,k)*exp(a1*qs(n,k)/240000) + I(n,k)*Ro;
    end
end


figure(6)
subplot(2,1,1)
hold all
plot([0:1:chargingTimeSec]/3600,v(:,1))
plot([0:1:chargingTimeSec]/3600,v(:,2))
plot([0:1:chargingTimeSec]/3600,v(:,3))
plot([0:1:chargingTimeSec]/3600,v(:,4))
plot([0:1:chargingTimeSec]/3600,v(:,5))
plot([0:1:chargingTimeSec]/3600,v(:,6))
plot([0:1:chargingTimeSec]/3600,v(:,7))
plot([0:1:chargingTimeSec]/3600,v(:,8))
plot([0:1:chargingTimeSec]/3600,v(:,9))
```

```
plot([0:1:chargingTimeSec]/3600,v(:,10))
plot([0:1:chargingTimeSec-1]/3600,4.2*ones(chargingTimeSec,1),'r')
plot([0:1:chargingTimeSec-1]/3600,2.5*ones(chargingTimeSec,1),'r')
xlabel('Time [Hours]')
ylabel('Voltage')

subplot(2,1,2)
hold all
plot([0:1:chargingTimeSec-1]/3600, I(:,1))
plot([0:1:chargingTimeSec-1]/3600, I(:,2))
plot([0:1:chargingTimeSec-1]/3600, I(:,3))
plot([0:1:chargingTimeSec-1]/3600, I(:,4))
plot([0:1:chargingTimeSec-1]/3600, I(:,5))
plot([0:1:chargingTimeSec-1]/3600, I(:,6))
plot([0:1:chargingTimeSec-1]/3600, I(:,7))
plot([0:1:chargingTimeSec-1]/3600, I(:,8))
plot([0:1:chargingTimeSec-1]/3600, I(:,9))
plot([0:1:chargingTimeSec-1]/3600, I(:,10))

xlabel('Time [Hours]')
ylabel('Current')
disp('Discharge cnt is: ')
disp(dischargeCnt)
```

APPENDIX B

ICMU CODE

```
/*****************************************************
This program was produced by the
CodeWizardAVR V2.04.8 Standard
Automatic Program Generator
 Copyright 1998-2010 Pavel Haiduc, HP InfoTech s.r.l.
http://www.hpinfotech.com

Project : Battery Board Slave
Version : .9
Date    : 10/15/2010
Author  :
Company : University of Akron
Comments:


Chip type               : ATxmega16A4
Program type            : Application
AVR Core Clock frequency: 32.000000 MHz
Memory model            : Small
Data Stack size         : 512
*****************************************************/

#include <io.h>
//#include <delay.h>
#include <math.h>
#include <stdio.h>
#include <avr_compiler.h>

#define MAX_AMP_SEC 216000.00
#define STOP_CHARGE_VOLT 3.8
#define OVERDISCHARGE_VOLT 2.5

//Equivalent to a voltage of 3.5V
```

```
#define DUTY_CYC_MAX 1330 //Limits Max Bypass Current to ~2 Amps at 3.8
Volts

//Current Integrating time in seconds
#define TSamp_TCD1 0.0039996

//Count to wait until MOSFET locks on after battVolt>3.8V mosCurrent>1.95A
and battCurr<1A
#define MAX_LOCK_MOS_CURRENT_CNT 700

//Count to wait until Duty Cycle Shuts off after seeing current out of battery
#define MAX_DUTY_OFF_CNT 700

//Number of bytes to be sent to MASTER
#define MAX_DATA_BYTES 10

//Address for all slave boards to respond to
#define allCallAddress 100

#define READ 1
#define WRITE 0

//LED Colors
#define GREEN 0b10
#define RED 0b01
#define AMBER 0b11

#define LOW_SOC 70 //Picked a value then the largest ledValue

#define IS_CHARGING_AT_2_MSG 0xF0
#define IS_NOT_CHARGING_AT_2_MSG 0xFA

static float battVolt;
static float battCurr;
static float mosCurr;
static float temp;
//static float prevBattCurr;
static float AmpSec=MAX_AMP_SEC;

signed long ADCtemp[4];
signed int ADCvalues[4];
char ADCsamplecount  = 0; // we're supersampling
signed int ADCbuffer;
char stepready = 0;
```

```
unsigned int voltCntrlCnt = 0;

unsigned char dataInTWIC=0;
unsigned char ledVal=0;

signed long OFFSET_CH0_TOTAL=0;
signed long OFFSET_CH1_TOTAL=0;
signed long OFFSET_CH2_TOTAL=0;

signed int OFFSET_CH0_DIFF=0;
signed int OFFSET_CH1_DIFF=0;
signed int OFFSET_CH2_DIFF=0;

//unsigned char lockMosCurr = 0;
unsigned int dutyOffCnt=0;

unsigned char battData[MAX_DATA_BYTES];

unsigned char dataByteCnt=0;

unsigned char slaveAddress = 0x00;
signed int offset=0;

unsigned char ledColor=GREEN;
unsigned char chargingAtTwoAmps=0;

//ADC OFFSET for each of the 10 boards OFFSET=M*RAWADC+B calibrated
manually
float const ADC_OFFSET_M[10]={-0.011107,-0.013504,-0.015376,-0.017009,-
0.015646,-0.013056,-0.015416,-0.015210,-0.013949,-0.012555};
float const ADC_OFFSET_B[10]={ 6.366141, 6.417454, 6.556141, 7.638119,
6.499410, 5.792097, 5.235612, 3.654257, 5.374430, 5.624966};


signed int bound(signed int value, signed int min, signed int max){
   if(value > max)
      return max;
   if(value < min)
      return min;
   return value;
}
```

```
#pragma warn-
char nvm_cmd_read( char *nvm_cmd_addr, char index ){
   #asm
      LDD  R30,Y+0    ; Z = index
      LDI  R31,0
       LDD  R26,Y+1    ; X = &NVM.CMD
       LDD  R27,Y+2
      LDI  R25,2      ; NVM.CMD = NVM_CMD_READ_CALIB_ROW_gc
      ST   X,R25
      LPM             ; read the data in R0
                  ; Clean up NVM Command register. */
      LDI  R25,0      ; NVM.CMD = NVM_CMD_NO_OPERATION_gc
      ST   X,R25
      MOV  R30,R0     ; return result
   #endasm
}
#pragma warn+

char SP_ReadCalibrationByte( char index ){
   return nvm_cmd_read(&NVM.CMD,index);
}


// System Clocks initialization
void system_clocks_init(void)
{
unsigned char n,s;

// Optimize for speed
#pragma optsize-
// Save interrupts enabled/disabled state
s=SREG;
// Disable interrupts
#asm("cli")

// Internal 32 kHz RC oscillator initialization
// Enable the internal 32 kHz RC oscillator
OSC.CTRL|=OSC_RC32KEN_bm;
// Wait for the internal 32 kHz RC oscillator to stabilize
while ((OSC.STATUS & OSC_RC32KRDY_bm)==0);

// Internal 32 MHz RC oscillator initialization
// Enable the internal 32 MHz RC oscillator
OSC.CTRL|=OSC_RC32MEN_bm;
```

```
// System Clock prescaler A division factor: 1
// System Clock prescalers B & C division factors: B:1, C:1
// ClkPer4: 32000.000 kHz
// ClkPer2: 32000.000 kHz
// ClkPer:  32000.000 kHz
// ClkCPU:  32000.000 kHz
n=(CLK.PSCTRL & (~(CLK_PSADIV_gm | CLK_PSBCDIV1_bm |
CLK_PSBCDIV0_bm))) |
   CLK_PSADIV_1_gc | CLK_PSBCDIV_1_1_gc;
CCP=CCP_IOREG_gc;
CLK.PSCTRL=n;

// Internal 32 MHz RC osc. calibration reference clock source: 32.768 kHz Internal
Osc.
OSC.DFLLCTRL&= ~(OSC_RC32MCREF_bm | OSC_RC2MCREF_bm);
// Enable the autocalibration of the internal 32 MHz RC oscillator
DFLLRC32M.CTRL|=DFLL_ENABLE_bm;

// Wait for the internal 32 MHz RC oscillator to stabilize
while ((OSC.STATUS & OSC_RC32MRDY_bm)==0);

// Select the system clock source: 32 MHz Internal RC Osc.
n=(CLK.CTRL & (~CLK_SCLKSEL_gm)) | CLK_SCLKSEL_RC32M_gc;
CCP=CCP_IOREG_gc;
CLK.CTRL=n;

// Disable the unused oscillators: 2 MHz, external clock/crystal oscillator, PLL
OSC.CTRL&= ~(OSC_RC2MEN_bm | OSC_XOSCEN_bm | OSC_PLLEN_bm);

// Peripheral Clock output: Disabled
PORTCFG.CLKEVOUT=(PORTCFG.CLKEVOUT &
(~PORTCFG_CLKOUT_gm)) | PORTCFG_CLKOUT_OFF_gc;

// Restore interrupts enabled/disabled state
SREG=s;

}

// Watchdog Timer initialization
void watchdog_init(void)
{
unsigned char s,n;
```

```
// Optimize for speed
#pragma optsize-
// Save interrupts enabled/disabled state
s=SREG;
// Disable interrupts
#asm("cli")

// Watchdog Timer: Off
n=(WDT.CTRL & (~WDT_ENABLE_bm)) | WDT_CEN_bm;
CCP=CCP_IOREG_gc;
WDT.CTRL=n;
// Watchdog window mode: Off
n=(WDT.WINCTRL & (~WDT_WEN_bm)) | WDT_WCEN_bm;
CCP=CCP_IOREG_gc;
WDT.WINCTRL=n;

// Restore interrupts enabled/disabled state
SREG=s;

}

// Event System initialization
void event_system_init(void)
{
// Event System Channel 0 source: Port D, Pin0
EVSYS.CH0MUX=EVSYS_CHMUX_PORTD_PIN0_gc;
// Event System Channel 1 source: Port D, Pin0
EVSYS.CH1MUX=EVSYS_CHMUX_OFF_gc;

// Event System Channel 2 source: None
EVSYS.CH2MUX=EVSYS_CHMUX_PORTD_PIN1_gc;
// Event System Channel 3 source: None
EVSYS.CH3MUX=EVSYS_CHMUX_OFF_gc;
// Event System Channel 4 source: None
EVSYS.CH4MUX=EVSYS_CHMUX_OFF_gc;
// Event System Channel 5 source: None
EVSYS.CH5MUX=EVSYS_CHMUX_OFF_gc;
// Event System Channel 6 source: None
EVSYS.CH6MUX=EVSYS_CHMUX_OFF_gc;
// Event System Channel 7 source: None
EVSYS.CH7MUX=EVSYS_CHMUX_OFF_gc;

// Event System Channel 0 Digital Filter Coefficient: 4 Samples
EVSYS.CH0CTRL=0b00001011;
```

```
// Event System Channel 1 Digital Filter Coefficient: 1 Sample
EVSYS.CH1CTRL=EVSYS_DIGFILT_1SAMPLE_gc;
// Event System Channel 2 Digital Filter Coefficient: 1 Sample
EVSYS.CH2CTRL=0b00001011;
// Event System Channel 3 Digital Filter Coefficient: 1 Sample
EVSYS.CH3CTRL=EVSYS_DIGFILT_1SAMPLE_gc;
// Event System Channel 4 Digital Filter Coefficient: 1 Sample
EVSYS.CH4CTRL=(EVSYS.CH4CTRL & (~(EVSYS_QDIRM_gm |
EVSYS_QDIEN_bm | EVSYS_QDEN_bm | EVSYS_DIGFILT_gm))) |
   EVSYS_DIGFILT_1SAMPLE_gc;
// Event System Channel 5 Digital Filter Coefficient: 1 Sample
EVSYS.CH5CTRL=EVSYS_DIGFILT_1SAMPLE_gc;
// Event System Channel 6 Digital Filter Coefficient: 1 Sample
EVSYS.CH6CTRL=EVSYS_DIGFILT_1SAMPLE_gc;
// Event System Channel 7 Digital Filter Coefficient: 1 Sample
EVSYS.CH7CTRL=EVSYS_DIGFILT_1SAMPLE_gc;

// Event System Channel 0 output: Disabled
// Note: the correct direction for the Event System Channel 0 output
// is configured in the ports_init function
PORTCFG.CLKEVOUT=(PORTCFG.CLKEVOUT &
(~PORTCFG_EVOUT_gm)) | PORTCFG_EVOUT_OFF_gc;

}

// Ports initialization
void ports_init(void)
{
// PORTA initialization
// OUT register
PORTA.OUT=0x00;

// Bit0: Input
// Bit1: Input
// Bit2: Input
// Bit3: Input
// Bit4: Input
// Bit5: Input
// Bit6: Input
// Bit7: Input
PORTA.DIR=0x00;

// Bit0 Output/Pull configuration: Totempole/No
// Bit0 Input/Sense configuration: Input buffer disabled
```

```
// Bit0 inverted: Off
// Bit0 slew rate limitation: Off
PORTA.PIN0CTRL=PORT_OPC_TOTEM_gc |
PORT_ISC_INPUT_DISABLE_gc;
// Bit1 Output/Pull configuration: Totempole/No
// Bit1 Input/Sense configuration: Input buffer disabled
// Bit1 inverted: Off
// Bit1 slew rate limitation: Off
PORTA.PIN1CTRL=PORT_OPC_TOTEM_gc |
PORT_ISC_INPUT_DISABLE_gc;
// Bit2 Output/Pull configuration: Totempole/No
// Bit2 Input/Sense configuration: Input buffer disabled
// Bit2 inverted: Off
// Bit2 slew rate limitation: Off
PORTA.PIN2CTRL=PORT_OPC_TOTEM_gc |
PORT_ISC_INPUT_DISABLE_gc;
// Bit3 Output/Pull configuration: Totempole/No
// Bit3 Input/Sense configuration: Input buffer disabled
// Bit3 inverted: Off
// Bit3 slew rate limitation: Off
PORTA.PIN3CTRL=PORT_OPC_TOTEM_gc |
PORT_ISC_INPUT_DISABLE_gc;
// Bit4 Output/Pull configuration: Totempole/No
// Bit4 Input/Sense configuration: Input buffer disabled
// Bit4 inverted: Off
// Bit4 slew rate limitation: Off
PORTA.PIN4CTRL=PORT_OPC_TOTEM_gc |
PORT_ISC_INPUT_DISABLE_gc;
// Bit5 Output/Pull configuration: Totempole/No
// Bit5 Input/Sense configuration: Input buffer disabled
// Bit5 inverted: Off
// Bit5 slew rate limitation: Off
PORTA.PIN5CTRL=PORT_OPC_TOTEM_gc |
PORT_ISC_INPUT_DISABLE_gc;
// Bit6 Output/Pull configuration: Totempole/No
// Bit6 Input/Sense configuration: Input buffer disabled
// Bit6 inverted: Off
// Bit6 slew rate limitation: Off
PORTA.PIN6CTRL=PORT_OPC_TOTEM_gc |
PORT_ISC_INPUT_DISABLE_gc;
// Bit7 Output/Pull configuration: Totempole/No
// Bit7 Input/Sense configuration: Input buffer disabled
// Bit7 inverted: Off
// Bit7 slew rate limitation: Off
```

```
PORTA.PIN7CTRL=PORT_OPC_TOTEM_gc |
PORT_ISC_INPUT_DISABLE_gc;
// Interrupt 0 level: Disabled
// Interrupt 1 level: Disabled
PORTA.INTCTRL=(PORTA.INTCTRL & (~(PORT_INT1LVL_gm |
PORT_INT0LVL_gm))) |
   PORT_INT1LVL_OFF_gc | PORT_INT0LVL_OFF_gc;
// Bit0 pin change interrupt 0: Off
// Bit1 pin change interrupt 0: Off
// Bit2 pin change interrupt 0: Off
// Bit3 pin change interrupt 0: Off
// Bit4 pin change interrupt 0: Off
// Bit5 pin change interrupt 0: Off
// Bit6 pin change interrupt 0: Off
// Bit7 pin change interrupt 0: Off
PORTA.INT0MASK=0x00;
// Bit0 pin change interrupt 1: Off
// Bit1 pin change interrupt 1: Off
// Bit2 pin change interrupt 1: Off
// Bit3 pin change interrupt 1: Off
// Bit4 pin change interrupt 1: Off
// Bit5 pin change interrupt 1: Off
// Bit6 pin change interrupt 1: Off
// Bit7 pin change interrupt 1: Off
PORTA.INT1MASK=0x00;

// PORTB initialization
// OUT register
PORTB.OUT=0x00;
// Bit0: Input
// Bit1: Input
// Bit2: Input
// Bit3: Input
PORTB.DIR=0x00;
// Bit0 Output/Pull configuration: Totempole/No
// Bit0 Input/Sense configuration: Input buffer disabled
// Bit0 inverted: Off
// Bit0 slew rate limitation: Off
PORTB.PIN0CTRL=PORT_OPC_TOTEM_gc |
PORT_ISC_INPUT_DISABLE_gc;
// Bit1 Output/Pull configuration: Totempole/No
// Bit1 Input/Sense configuration: Input buffer disabled
// Bit1 inverted: Off
// Bit1 slew rate limitation: Off
```

```
PORTB.PIN1CTRL=PORT_OPC_TOTEM_gc |
PORT_ISC_INPUT_DISABLE_gc;
// Bit2 Output/Pull configuration: Totempole/No
// Bit2 Input/Sense configuration: Input buffer disabled
// Bit2 inverted: Off
// Bit2 slew rate limitation: Off
PORTB.PIN2CTRL=PORT_OPC_PULLUP_gc |
PORT_ISC_INPUT_DISABLE_gc;
// Bit3 Output/Pull configuration: Totempole/No
// Bit3 Input/Sense configuration: Input buffer disabled
// Bit3 inverted: Off
// Bit3 slew rate limitation: Off
PORTB.PIN3CTRL=PORT_OPC_PULLUP_gc |
PORT_ISC_INPUT_DISABLE_gc;
// Interrupt 0 level: Disabled
// Interrupt 1 level: Disabled
PORTB.INTCTRL=(PORTB.INTCTRL & (~(PORT_INT1LVL_gm |
PORT_INT0LVL_gm))) |
   PORT_INT1LVL_OFF_gc | PORT_INT0LVL_OFF_gc;
// Bit0 pin change interrupt 0: Off
// Bit1 pin change interrupt 0: Off
// Bit2 pin change interrupt 0: Off
// Bit3 pin change interrupt 0: Off
PORTB.INT0MASK=0x00;
// Bit0 pin change interrupt 1: Off
// Bit1 pin change interrupt 1: Off
// Bit2 pin change interrupt 1: Off
// Bit3 pin change interrupt 1: Off
PORTB.INT1MASK=0x00;

// PORTC initialization
// OUT register
PORTC.OUT=0x00;

// Bit0: Output
// Bit1: Output
// Bit2: Input
// Bit3: Input
// Bit4: Output
// Bit5: Input
// Bit6: Input
// Bit7: Output
PORTC.DIR=0x93;
```

```
// Bit0 Output/Pull configuration: WIRED-AND (on input)
// Bit0 Input/Sense configuration: Sense both edges
// Bit0 inverted: Off
// Bit0 slew rate limitation: Off
PORTC.PIN0CTRL=PORT_OPC_WIREDAND_gc |
PORT_ISC_BOTHEDGES_gc;
// Bit1 Output/Pull configuration: WIRED-AND (on input)
// Bit1 Input/Sense configuration: Sense both edges
// Bit1 inverted: Off
// Bit1 slew rate limitation: Off
PORTC.PIN1CTRL=PORT_OPC_WIREDAND_gc |
PORT_ISC_BOTHEDGES_gc;
// Bit2 Output/Pull configuration: Totempole/No
// Bit2 Input/Sense configuration: Sense both edges
// Bit2 inverted: Off
// Bit2 slew rate limitation: Off
PORTC.PIN2CTRL=PORT_OPC_TOTEM_gc | PORT_ISC_BOTHEDGES_gc;
// Bit3 Output/Pull configuration: Totempole/No
// Bit3 Input/Sense configuration: Sense both edges
// Bit3 inverted: Off
// Bit3 slew rate limitation: Off
PORTC.PIN3CTRL=PORT_OPC_TOTEM_gc | PORT_ISC_BOTHEDGES_gc;
// Bit4 Output/Pull configuration: Totempole/Pull-up (on input)
// Bit4 Input/Sense configuration: Sense both edges
// Bit4 inverted: Off
// Bit4 slew rate limitation: Off
PORTC.PIN4CTRL=PORT_OPC_PULLDOWN_gc |
PORT_ISC_BOTHEDGES_gc;
// Bit5 Output/Pull configuration: Totempole/Pull-up (on input)
// Bit5 Input/Sense configuration: Sense both edges
// Bit5 inverted: Off
// Bit5 slew rate limitation: Off
PORTC.PIN5CTRL=PORT_OPC_PULLUP_gc | PORT_ISC_BOTHEDGES_gc;
// Bit6 Output/Pull configuration: Totempole/Pull-up (on input)
// Bit6 Input/Sense configuration: Sense both edges
// Bit6 inverted: Off
// Bit6 slew rate limitation: Off
PORTC.PIN6CTRL=PORT_OPC_PULLUP_gc | PORT_ISC_BOTHEDGES_gc;
// Bit7 Output/Pull configuration: Totempole/Pull-up (on input)
// Bit7 Input/Sense configuration: Sense both edges
// Bit7 inverted: Off
// Bit7 slew rate limitation: Off
PORTC.PIN7CTRL=PORT_OPC_TOTEM_gc | PORT_ISC_BOTHEDGES_gc;
// Interrupt 0 level: Disabled
```

```
// Interrupt 1 level: Disabled
PORTC.INTCTRL=(PORTC.INTCTRL & (~(PORT_INT1LVL_gm |
PORT_INT0LVL_gm))) |
   PORT_INT1LVL_OFF_gc | PORT_INT0LVL_OFF_gc;
// Bit0 pin change interrupt 0: Off
// Bit1 pin change interrupt 0: Off
// Bit2 pin change interrupt 0: Off
// Bit3 pin change interrupt 0: Off
// Bit4 pin change interrupt 0: Off
// Bit5 pin change interrupt 0: Off
// Bit6 pin change interrupt 0: Off
// Bit7 pin change interrupt 0: Off
PORTC.INT0MASK=0x00;
// Bit0 pin change interrupt 1: Off
// Bit1 pin change interrupt 1: Off
// Bit2 pin change interrupt 1: Off
// Bit3 pin change interrupt 1: Off
// Bit4 pin change interrupt 1: Off
// Bit5 pin change interrupt 1: Off
// Bit6 pin change interrupt 1: Off
// Bit7 pin change interrupt 1: Off
PORTC.INT1MASK=0x00;

// PORTD initialization
// OUT register
PORTD.OUT=0x00;

// Bit0: Output
// Bit1: Output
// Bit2: Output
// Bit3: Output
// Bit4: Output
// Bit5: Output
// Bit6: Output
// Bit7: Output
PORTD.DIR=0xFF;

// Bit0 Output/Pull configuration: Totempole/Pull-up (on input)
// Bit0 Input/Sense configuration: Sense low level
// Bit0 inverted: Off
// Bit0 slew rate limitation: Off
PORTD.PIN0CTRL=PORT_OPC_TOTEM_gc | PORT_ISC_LEVEL_gc;

// Bit1 Output/Pull configuration: Totempole/Pull-up (on input)
```

```
// Bit1 Input/Sense configuration: Sense low level
// Bit1 inverted: Off
// Bit1 slew rate limitation: Off
PORTD.PIN1CTRL=PORT_OPC_TOTEM_gc | PORT_ISC_LEVEL_gc;
// Bit2 Output/Pull configuration: Totempole/Pull-up (on input)
// Bit2 Input/Sense configuration: Sense both edges
// Bit2 inverted: Off
// Bit2 slew rate limitation: Off
PORTD.PIN2CTRL=PORT_OPC_TOTEM_gc | PORT_ISC_BOTHEDGES_gc;
// Bit3 Output/Pull configuration: Totempole/Pull-up (on input)
// Bit3 Input/Sense configuration: Sense both edges
// Bit3 inverted: Off
// Bit3 slew rate limitation: Off
PORTD.PIN3CTRL=PORT_OPC_TOTEM_gc | PORT_ISC_BOTHEDGES_gc;
// Bit4 Output/Pull configuration: Totempole/Pull-up (on input)
// Bit4 Input/Sense configuration: Sense both edges
// Bit4 inverted: Off
// Bit4 slew rate limitation: Off
PORTD.PIN4CTRL=PORT_OPC_TOTEM_gc | PORT_ISC_BOTHEDGES_gc;
// Bit5 Output/Pull configuration: Totempole/Pull-up (on input)
// Bit5 Input/Sense configuration: Sense both edges
// Bit5 inverted: Off
// Bit5 slew rate limitation: Off
PORTD.PIN5CTRL=PORT_OPC_TOTEM_gc | PORT_ISC_BOTHEDGES_gc;
// Bit6 Output/Pull configuration: Totempole/No
// Bit6 Input/Sense configuration: Sense both edges
// Bit6 inverted: Off
// Bit6 slew rate limitation: Off
PORTD.PIN6CTRL=PORT_OPC_TOTEM_gc | PORT_ISC_BOTHEDGES_gc;
// Bit7 Output/Pull configuration: Totempole/No
// Bit7 Input/Sense configuration: Sense both edges
// Bit7 inverted: Off
// Bit7 slew rate limitation: Off
PORTD.PIN7CTRL=PORT_OPC_TOTEM_gc | PORT_ISC_BOTHEDGES_gc;
// Interrupt 0 level: Disabled
// Interrupt 1 level: Disabled

PORTD.INTCTRL=(PORTD.INTCTRL & (~(PORT_INT1LVL_gm |
PORT_INT0LVL_gm))) |
   PORT_INT1LVL_OFF_gc | PORT_INT0LVL_OFF_gc;
// Bit0 pin change interrupt 0: Off
// Bit1 pin change interrupt 0: Off
// Bit2 pin change interrupt 0: Off
// Bit3 pin change interrupt 0: Off
```

```
// Bit4 pin change interrupt 0: Off
// Bit5 pin change interrupt 0: Off
// Bit6 pin change interrupt 0: Off
// Bit7 pin change interrupt 0: Off

PORTD.INT0MASK=0x00;
// Bit0 pin change interrupt 1: Off
// Bit1 pin change interrupt 1: Off
// Bit2 pin change interrupt 1: Off
// Bit3 pin change interrupt 1: Off
// Bit4 pin change interrupt 1: Off
// Bit5 pin change interrupt 1: Off
// Bit6 pin change interrupt 1: Off
// Bit7 pin change interrupt 1: Off
PORTD.INT1MASK=0x00;

// PORTE initialization
// OUT register
PORTE.OUT=0x00; //Initial Output Value

// Bit0: Output
// Bit1: Output
// Bit2: Output
// Bit3: Output
PORTE.DIR=0x0F;

// Bit0 Output/Pull configuration: Totempole/No
// Bit0 Input/Sense configuration: Sense both edges
// Bit0 inverted: Off
// Bit0 slew rate limitation: Off
PORTE.PIN0CTRL=PORT_OPC_TOTEM_gc | PORT_ISC_BOTHEDGES_gc;

// Bit1 Output/Pull configuration: Totempole/No
// Bit1 Input/Sense configuration: Sense both edges
// Bit1 inverted: On
// Bit1 slew rate limitation: Off
PORTE.PIN1CTRL=PORT_OPC_TOTEM_gc | PORT_ISC_BOTHEDGES_gc;

// Bit2 Output/Pull configuration: Totempole/Pull-up (on input)
// Bit2 Input/Sense configuration: Sense both edges
// Bit2 inverted: Off
// Bit2 slew rate limitation: Off
PORTE.PIN2CTRL=PORT_OPC_TOTEM_gc | PORT_ISC_BOTHEDGES_gc;
```

```
// Bit3 Output/Pull configuration: Totempole/Pull-up (on input)
// Bit3 Input/Sense configuration: Sense both edges
// Bit3 inverted: Off
// Bit3 slew rate limitation: Off
PORTE.PIN3CTRL=PORT_OPC_TOTEM_gc | PORT_ISC_BOTHEDGES_gc;
// Interrupt 0 level: Disabled
// Interrupt 1 level: Disabled

PORTE.INTCTRL=(PORTE.INTCTRL & (~(PORT_INT1LVL_gm |
PORT_INT0LVL_gm))) |
   PORT_INT1LVL_OFF_gc | PORT_INT0LVL_OFF_gc;
// Bit0 pin change interrupt 0: Off
// Bit1 pin change interrupt 0: Off
// Bit2 pin change interrupt 0: Off
// Bit3 pin change interrupt 0: Off
PORTE.INT0MASK=0x00;
// Bit0 pin change interrupt 1: Off
// Bit1 pin change interrupt 1: Off
// Bit2 pin change interrupt 1: Off
// Bit3 pin change interrupt 1: Off
PORTE.INT1MASK=0x00;

// PORTR initialization
// OUT register
PORTR.OUT=0x00;
// Bit0: Input
// Bit1: Input
PORTR.DIR=0x00;
// Bit0 Output/Pull configuration: Totempole/No
// Bit0 Input/Sense configuration: Sense both edges
// Bit0 inverted: Off
// Bit0 slew rate limitation: Off
PORTR.PIN0CTRL=PORT_OPC_PULLUP_gc | PORT_ISC_BOTHEDGES_gc;
// Bit1 Output/Pull configuration: Totempole/No
// Bit1 Input/Sense configuration: Sense both edges
// Bit1 inverted: Off
// Bit1 slew rate limitation: Off
PORTR.PIN1CTRL=PORT_OPC_PULLUP_gc | PORT_ISC_BOTHEDGES_gc;
// Interrupt 0 level: Disabled
// Interrupt 1 level: Disabled
PORTR.INTCTRL=(PORTR.INTCTRL & (~(PORT_INT1LVL_gm |
PORT_INT0LVL_gm))) |
   PORT_INT1LVL_OFF_gc | PORT_INT0LVL_OFF_gc;
// Bit0 pin change interrupt 0: Off
```

```
// Bit1 pin change interrupt 0: Off
PORTR.INT0MASK=0x00;
// Bit0 pin change interrupt 1: Off
// Bit1 pin change interrupt 1: Off
PORTR.INT1MASK=0x00;
}

/*
// Virtual Ports initialization
void vports_init(void)
{
// PORTA mapped to VPORT0
// PORTB mapped to VPORT1
PORTCFG.VPCTRLA=PORTCFG_VP1MAP_PORTB_gc |
PORTCFG_VP0MAP_PORTA_gc;
// PORTC mapped to VPORT2
// PORTD mapped to VPORT3
PORTCFG.VPCTRLB=PORTCFG_VP3MAP_PORTD_gc |
PORTCFG_VP2MAP_PORTC_gc;
}
*/

// Disable a Timer/Counter type 0
void tc0_disable(TC0_t *ptc)
{
// Timer/Counter off
ptc->CTRLA=(ptc->CTRLA & (~TC0_CLKSEL_gm)) | TC_CLKSEL_OFF_gc;
// Issue a reset command
ptc->CTRLFSET=TC_CMD_RESET_gc;
}

// Disable a Timer/Counter type 1
void tc1_disable(TC1_t *ptc)
{
// Timer/Counter off
ptc->CTRLA=(ptc->CTRLA & (~TC1_CLKSEL_gm)) | TC_CLKSEL_OFF_gc;
// Issue a reset command
ptc->CTRLFSET=TC_CMD_RESET_gc;
}



// Timer/counter TCC1 Overflow/Underflow interrupt service routine
interrupt [TCC0_OVF_vect] void tcc0_overflow_isr(void)
```

```
{
// write your code here

}


// Timer/Counter TCD0 initialization
void tcd0_init(void)
{
unsigned char s;

// Note: the correct PORTD direction for the Compare Channels outputs
// is configured in the ports_init function

// Save interrupts enabled/disabled state
s=SREG;
// Disable interrupts
#asm("cli")

// Disable and reset the timer/counter just to be sure
tc0_disable(&TCD0);
// Clock source: Peripheral Clock/1
TCD0.CTRLA=0;
TCD0.CTRLB=0b00000000;
TCD0.CTRLC=0b00000000;
TCD0.CTRLD=0b01101010;
TCD0.CTRLE=0b00000000;

// Overflow interrupt: Medium Level
// Error interrupt: Disabled
TCD0.INTCTRLA=(TCD0.INTCTRLA & (~(TC0_ERRINTLVL_gm |
TC0_OVFINTLVL_gm))) |
   TC_ERRINTLVL_OFF_gc | TC_OVFINTLVL_MED_gc;


// Compare/Capture channel A interrupt: Disabled
// Compare/Capture channel B interrupt: Disabled
// Compare/Capture channel C interrupt: Disabled
// Compare/Capture channel D interrupt: Disabled
TCD0.INTCTRLB=(TCD0.INTCTRLB & (~(TC0_CCDINTLVL_gm |
TC0_CCCINTLVL_gm | TC0_CCBINTLVL_gm | TC0_CCAINTLVL_gm))) |
   TC_CCDINTLVL_OFF_gc | TC_CCCINTLVL_OFF_gc |
TC_CCBINTLVL_OFF_gc | TC_CCAINTLVL_OFF_gc;
```

```
// High resolution extension: Off
HIRESD.CTRL&= ~HIRES_HREN0_bm;

// Clear the interrupt flags
TCD0.INTFLAGS=TCD0.INTFLAGS;
// Set counter register
TCD0.CNT=0x0000;
// Set period register
TCD0.PER=20719;
// Set channel A Compare/Capture register
TCD0.CCA=0x0000;
// Set channel B Compare/Capture register
TCD0.CCB=0x0000;
// Set channel C Compare/Capture register
TCD0.CCC=0x0000;
// Set channel D Compare/Capture register
TCD0.CCD=0x0000;

// Restore interrupts enabled/disabled state
SREG=s;

TCD0.CTRLA=0b00000001;
}


// Timer/counter TCD0 Overflow/Underflow interrupt service routine
interrupt [TCD0_OVF_vect] void tcd0_overflow_isr(void)
{

}



// Timer/Counter TCD1 initialization
void tcd1_init(void)
{
unsigned char s;

// Note: the correct PORTD direction for the Compare Channels outputs
// is configured in the ports_init function

// Save interrupts enabled/disabled state
s=SREG;
// Disable interrupts
```

```
#asm("cli")

// Disable and reset the timer/counter just to be sure
tc1_disable(&TCD1);
// Clock source: Peripheral Clock/2
TCD1.CTRLA=(TCD1.CTRLA & (~TC1_CLKSEL_gm)) |
TC_CLKSEL_DIV2_gc;

// Mode: Normal Operation, Overflow Int./Event on TOP
// Compare/Capture on channel A: Off
// Compare/Capture on channel B: Off
TCD1.CTRLB=(TCD1.CTRLB & (~(TC1_CCAEN_bm | TC1_CCBEN_bm |
TC1_WGMODE_gm))) |
   TC_WGMODE_NORMAL_gc;

// Capture event source: None
// Capture event action: None
TCD1.CTRLD=(TCD1.CTRLD & (~(TC1_EVACT_gm | TC1_EVSEL_gm))) |
   TC_EVACT_OFF_gc | TC_EVSEL_OFF_gc;

// Overflow interrupt: Low Level
// Error interrupt: Enabled
TCD1.INTCTRLA=(TCD1.INTCTRLA & (~(TC1_ERRINTLVL_gm ))) |
   TC_ERRINTLVL_OFF_gc | TC_OVFINTLVL_LO_gc |
TC1_OVFINTLVL_gm;

// Compare/Capture channel A interrupt: Disabled
// Compare/Capture channel B interrupt: Disabled
TCD1.INTCTRLB=(TCD1.INTCTRLB & (~(TC1_CCBINTLVL_gm |
TC1_CCAINTLVL_gm))) |
   TC_CCBINTLVL_OFF_gc | TC_CCAINTLVL_OFF_gc;

// High resolution extension: Off
HIRESD.CTRL&= ~HIRES_HREN1_bm;

// Clear the interrupt flags
TCD1.INTFLAGS=TCD1.INTFLAGS;
// Set counter register
TCD1.CNT=0x0000;
// Set period register
TCD1.PER=64000;
// Set channel A Compare/Capture register
TCD1.CCA=0x0000;
// Set channel B Compare/Capture register
```

```
TCD1.CCB=0x0000;

// Restore interrupts enabled/disabled state
SREG=s;
}

// Timer/counter TCD1 Overflow/Underflow interrupt service routine
interrupt [TCD1_OVF_vect] void tcd1_overflow_isr(void){
   stepready = 1;

   if (battCurr>0.1 || battCurr<-0.35){
      AmpSec = AmpSec + battCurr*TSamp_TCD1;
   }else if( battCurr>-0.35 && battCurr<0.0){
      AmpSec = AmpSec + (battCurr+.05)*TSamp_TCD1;
   }

   if(AmpSec<=0) AmpSec=0;
   else if(AmpSec>MAX_AMP_SEC) AmpSec=MAX_AMP_SEC;

}




// Timer/Counter TCE0 initialization
void tcc1_init(void)
{
unsigned char s;

// Note: the correct PORTC direction for the Compare Channels outputs
// is configured in the ports_init function

// Save interrupts enabled/disabled state
s=SREG;
// Disable interrupts
#asm("cli")

// Disable and reset the timer/counter just to be sure
tc1_disable(&TCC1);
// Clock source: Peripheral Clock/1
TCC1.CTRLA=(TCC1.CTRLA & (~TC1_CLKSEL_gm)) |
TC_CLKSEL_DIV1_gc;

// Mode: Dual Slope PWM Gen., Overflow Int./Event on TOP & BOTTOM
```

```
// Compare/Capture on channel A: Off
// Compare/Capture on channel B: On
// Compare/Capture on channel C: Off
// Compare/Capture on channel D: Off
//TCC0.CTRLB=(TCC0.CTRLB & (~(TC0_CCAEN_bm | TC0_CCBEN_bm |
TC0_CCCEN_bm | TC0_CCDEN_bm | TC0_WGMODE_gm))) |
//    TC0_CCBEN_bm | TC0_CCAEN_bm|
//    TC_WGMODE_DS_TB_gc;

TCC1.CTRLB = 0b00000011; //W
TCC1.CTRLB = TCC1.CTRLB | 0b00010000;


// Capture event source: None
// Capture event action: None
TCC1.CTRLD=(TCC0.CTRLD & (~(TC0_EVACT_gm | TC0_EVSEL_gm))) |
TC_EVACT_OFF_gc | TC_EVSEL_OFF_gc;

// Overflow interrupt: Medium Level
// Error interrupt: Disabled
// TCC0.INTCTRLA=(TCC0.INTCTRLA & (~(TC0_ERRINTLVL_gm |
TC0_OVFINTLVL_gm))) |
//    TC_ERRINTLVL_OFF_gc | TC_OVFINTLVL_MED_gc;

// Compare/Capture channel A interrupt: Disabled
// Compare/Capture channel B interrupt: Disabled

//TCC0.INTCTRLB=(TCC0.INTCTRLB & (~(TC0_CCDINTLVL_gm |
TC0_CCCINTLVL_gm | TC0_CCBINTLVL_gm | TC0_CCAINTLVL_gm))) |
//    TC_CCDINTLVL_OFF_gc | TC_CCCINTLVL_OFF_gc |
TC_CCBINTLVL_OFF_gc | TC_CCAINTLVL_OFF_gc;

// High resolution extension: Off
HIRESC.CTRL&= ~HIRES_HREN0_bm;

// Clear the interrupt flags
TCC1.INTFLAGS=0;

// Set counter value
TCC1.CNT=0x0000;

// Set period register
TCC1.PER=1600-1;
```

```
// Set channel A Compare/Capture register
TCC1.CCA=0;
// Set channel B Compare/Capture register
TCC1.CCB=0x0000;

// Restore interrupts enabled/disabled state
SREG=s;
}

// Disable an USART
void usart_disable(USART_t *pu)
{
// Rx and Tx are off
pu->CTRLB=0;
// Ensure that all interrupts generated by the USART are off
pu->CTRLA=0;
}

void adc_init(void){

   ADCA.CALL = SP_ReadCalibrationByte( PROD_SIGNATURES_START +
ADCACAL0_offset );
   ADCA.CALH = SP_ReadCalibrationByte( PROD_SIGNATURES_START +
ADCACAL1_offset );


   ADCA.CTRLA = 0b00111101; // set up with a four channel sweep
   ADCA.CTRLB = 0b00010000; // signed
   ADCA.REFCTRL =0b00110000;     //portb reference, temp and bandgap
disabled
   ADCA.EVCTRL = 0b11000000;  // no events
   ADCA.PRESCALER = 0b00000101; // div 128, 250kHz
   //ADCA.PRESCALER = 0b00000110; // div 256, 125kHz
   //ADCA.PRESCALER = 0b00000011; // div 32
   //ADCA.PRESCALER = 0b00000111; // div 512, 62.5kHz

   ADCA.CH0.CTRL =0b10000011; // start channel 0, differential w/gain 1
   ADCA.CH0.MUXCTRL = 0b00000011; // Differential POS=A0 and NEG=A7,
Battery Voltage
   ADCA.CH0.INTCTRL = 0b00000000; // no interrupt on channel 0;

   //ADCA.CH1.CTRL =0b10011011; // start channel 1, differential w/gain 64
   //ADCA.CH1.CTRL =0b10010011; // start channel 1, differential w/gain 16
   ADCA.CH1.CTRL =0b10001111; // start channel 1, differential w/gain 8
```

```
    ADCA.CH1.MUXCTRL = 0b00010010;  // Differential POS=A2 and NEG=A6,
Battery Current
    //ADCA.CH1.MUXCTRL = 0b00110010;  // Differential POS=A6 and
NEG=A6, Battery Current
    ADCA.CH1.INTCTRL = 0b00000000; // no interrupt on channel 1;

    //ADCA.CH2.CTRL = 0b10011011; // start channel 2, differential w/gain 64
    ADCA.CH2.CTRL = 0b10000011;   // start channel 2, differential w/gain 1
    ADCA.CH2.MUXCTRL = 0b00100001; // Differential POS=A4 and NEG=A5,
MOSFET Current
    //ADCA.CH2.CTRL = 0b10000011;
    //ADCA.CH2.MUXCTRL = 0b00101001; // Differential POS=A4 and
NEG=A5, MOSFET Current
    ADCA.CH2.INTCTRL = 0b00000000; // no interrupt on channel 2;

    ADCA.CH3.CTRL = 0b10000001; // start channel 3, single-ended
    ADCA.CH3.MUXCTRL = 0b00011000; // Batt temp
    ADCA.CH3.INTCTRL = 0b00000001; // low interrupt on channel 3;
}

// ADC interrupt service routine
interrupt [ADCA_CH3_vect] void ADCA_CH3_isr(void){
    char i;
    // the ADC does not seem to bounds check properly, so I'll have to do it
    ADCbuffer = ADCA.CH0.RES-OFFSET_CH0_DIFF;
    ADCtemp[0] += bound(ADCbuffer, 0, 2047);

    ADCbuffer = ADCA.CH1.RES-OFFSET_CH1_DIFF;
    ADCtemp[1] += bound(ADCbuffer, -2048, 2047);

    ADCbuffer = ADCA.CH2.RES-OFFSET_CH2_DIFF;
    ADCtemp[2] += bound(ADCbuffer, -2048, 2047);

    ADCbuffer = ADCA.CH3.RES;
    ADCtemp[3] += bound(ADCbuffer, 0, 2047);

    ADCsamplecount++; //16 super sample
    if(ADCsamplecount > 16){
        //store the values
        ADCvalues[0] = ADCtemp[0]>>4;
        ADCvalues[1] = ADCtemp[1]>>4;
        ADCvalues[2] = ADCtemp[2]>>4;
        ADCvalues[3] = ADCtemp[3]>>4;
```

```
      for(i = 0; i<4; i++){
         ADCtemp[i] = 0;
      }

      ADCsamplecount = 0;
   }

   //Initiate next samples
   ADCA.CH0.CTRL |=0b10000000;
   ADCA.CH1.CTRL |=0b10000000;
   ADCA.CH2.CTRL |=0b10000000;
   ADCA.CH3.CTRL |=0b10000000;
}


interrupt [TWIC_TWIS_vect] void TWIC_TWIS_isr(void){

     if ((TWIC.SLAVE.STATUS &
TWI_SLAVE_APIF_bm)>>TWI_SLAVE_APIF_bp){
        //if ((TWIC.SLAVE.STATUS &
TWI_SLAVE_DIR_bm)>>TWI_SLAVE_DIR_bp){ //CASE M1
           //MASTER READ OPERATION, SEND ACK
            TWIC.SLAVE.CTRLB = (TWIC.SLAVE.CTRLB &
~TWI_SLAVE_CMD_gm & ~TWI_SLAVE_ACKACT_bm) |
TWI_SLAVE_CMD_RESPONSE_gc;
            dataByteCnt=0;
     }else if ((TWIC.SLAVE.STATUS &
TWI_SLAVE_DIF_bm)>>TWI_SLAVE_DIF_bp){

        if ((TWIC.SLAVE.STATUS &
TWI_SLAVE_DIR_bm)>>TWI_SLAVE_DIR_bp){
           if(dataByteCnt<MAX_DATA_BYTES){
             TWIC.SLAVE.DATA =  battData[dataByteCnt++];
             TWIC.SLAVE.STATUS |= TWI_SLAVE_DIF_bm;
           }else{
             //Automatically clears DIF with command being sent
             TWIC.SLAVE.CTRLB = (TWIC.SLAVE.CTRLB &
~TWI_SLAVE_CMD_gm & ~TWI_SLAVE_ACKACT_bm) |
TWI_SLAVE_CMD_COMPTRANS_gc | TWI_SLAVE_ACKACT_bm;
             PORTE.OUTCLR = 0b00001010;
             PORTD.OUTCLR = 0b10101010;
           }
        }else{
          //MASTER WRITE OPERATION
```

```
            dataInTWIC = TWIC.SLAVE.DATA;
            if(dataInTWIC == LOW_SOC){
               ledVal=LOW_SOC;
            }else{
               if(dataInTWIC>slaveAddress*6){
                  ledVal=7;
               }else{
                  ledVal=dataInTWIC-(slaveAddress-1)*6;
                  if(ledVal>6) ledVal=7;
               }
            }
            TWIC.SLAVE.CTRLB = (TWIC.SLAVE.CTRLB &
~TWI_SLAVE_CMD_gm & ~TWI_SLAVE_ACKACT_bm) |
TWI_SLAVE_CMD_COMPTRANS_gc;
         }
      }
}


void init_I2C_Slave(){

   //BIT 7:3 RESERVED
   //BIT 2, ACKACT, 0=SEND ACK, 1= SEND NACK
   //BIT 1:0, cmd
   TWIC.SLAVE.CTRLB = 0b00000000;

   //BIT 7 DIF: Data Interrupt Flag
   //BIT 6 APIF: Address/Stop Interrupt Flag
   //BIT 5 CLKHOLD: Clk Hold Flag
   //BIT 4 RXACK: Received Ack flag
   //BIT 3 COLL: collision
   //BIT 2 BUSERR: Twi slave bus error
   //BIT 1 DIR: Read/Write Direction bit 1=Master Read, 0=Write
   //BIT 0 Slave Address or Stop Flag WHY APIF WAS SET, 0=STOP
1=ADDRESS
   TWIC.SLAVE.STATUS = 0b11001100;

   TWIC.SLAVE.ADDR = slaveAddress<<1;
   TWIC.SLAVE.DATA = 0b00000000;

   //Mask Off
   TWIC.SLAVE.ADDRMASK = allCallAddress<<1|1; //Second address all
slaves listen for messages on
```

```
    //BIT 7:6 Slave interrupt level?
    //BIT 5, DIE Data interrupt enable, 1=ON
    //BIT 4, APIEN, Address/Stop Interrupt Enable, 1=ON
    //BIT 3, ENABLE TWI SLAVE, 1=ON
    //BIT 2, PIEN, Stop Interrupt Enable, 1=ON
    //Bit 1, PMEN, Promiscuous Mode Enable, 0=OFF
    //BIT 0, SMEN, Smart mode enable, ?
    TWIC.SLAVE.CTRLA = 0b01111000;


}

void main(void)
{

// Declare your local variables here
unsigned char byPassOn=0;
unsigned char byPassMax=0;
unsigned char overTemp=0;
unsigned char overDischarge=0;


int movAvgValues[3][64];
long movAvgTotal[3]={0,0,0};
int newAdcVal[3];
unsigned int initSamples=0;
unsigned int sampleIdxOldest=0;

unsigned char n;


int mosCurrRaw;
int battCurrRaw;
int battVoltRaw;
unsigned int socRaw=0x0700;
unsigned int tempRaw=0;

unsigned int i=0;

unsigned int duty=0;

float error = 0.0;
float refV = 0.0;
float KpV = 0;
```

```
//float KpI = 0;
//float refI = 0.0;

//float Ki = 0;
//float integral_I = 0;
float integral_V = 0;
//float T = 0;


// Make sure the interrupts are disabled
#asm("cli")
// Low level interrupt: On
// Round-robin scheduling for low level interrupt: Off
// Medium level interrupt: On
// High level interrupt: On
// The interrupt vectors will be placed at the start of the Application FLASH section
n=(PMIC.CTRL & (~(PMIC_RREN_bm | PMIC_IVSEL_bm |
PMIC_HILVLEN_bm | PMIC_MEDLVLEN_bm | PMIC_LOLVLEN_bm))) |
   PMIC_LOLVLEN_bm | PMIC_MEDLVLEN_bm | PMIC_HILVLEN_bm;
CCP=CCP_IOREG_gc;
PMIC.CTRL=n;
// Set the default priority for round-robin scheduling
PMIC.INTPRI=0x00;

// Watchdog timer initialization
watchdog_init();

// System clocks initialization
system_clocks_init();

// Event system initialization
event_system_init();

// Virtual Ports initialization
//vports_init();

ports_init();

delay_ms(1000);
// Timer/Counter TCC1 initialization
tcc1_init();

// Timer/Counter TCD0 is disabled
//tcd0_init();
```

```
// Timer/Counter TCD1 is enabled
tcd1_init();

// Timer/Counter TCE0 initialization
//tce0_init();

// RTC initialization
//rtcxm_init();

// USARTC0 is disabled
usart_disable(&USARTC0);

// USARTC1 is disabled
usart_disable(&USARTC1);

// USARTD0 is disabled
usart_disable(&USARTD0);

// USARTD1 is disabled
usart_disable(&USARTD1);

// USARTE0 is disabled
usart_disable(&USARTE0);

// ADC Initilization
adc_init();
//usart_init();


delay_ms(1000);
//Get Address for slave from hardware pins (they were shorted)
//PB2 = BIT 0
//PB3 = BIT 1
//PR0 = BIT 2
//PR1 = BIT 3
slaveAddress=0;
if((PORTB.IN&PIN2_bm)>>PIN2_bp) slaveAddress |= 1;
if((PORTB.IN&PIN3_bm)>>PIN3_bp) slaveAddress |= (1<<1);
if((PORTR.IN&PIN0_bm)>>PIN0_bp) slaveAddress |= (1<<2);
if((PORTR.IN&PIN1_bm)>>PIN1_bp) slaveAddress |= (1<<3);
slaveAddress = slaveAddress+1; //Offset by 1 since addresses cannot be 0

for(i=0;i<slaveAddress;i++){
```

```
   PORTD.OUTTGL = 0b00000010;
   delay_ms(200);
   PORTD.OUTTGL = 0b00000010;
   delay_ms(200);
}


//Initialize Slave
init_I2C_Slave();

ADCA.CH0.MUXCTRL = 0b00110010;  // Differential POS=A6 and NEG=A6
ADCA.CH1.MUXCTRL = 0b00110010;  // Differential POS=A6 and NEG=A6
ADCA.CH2.MUXCTRL = 0b00110010;  // Differential POS=A6 and NEG=A6

//Calibrate Differential Gain Offset
for (i=0; i<64; i++){
   ADCA.CH0.CTRL |=0b10000000;
   ADCA.CH1.CTRL |=0b10000000;
   ADCA.CH2.CTRL |=0b10000000;
   while((ADCA.CH2.INTFLAGS&0x01)!=1);
   ADCA.CH2.INTFLAGS |= 0x01;  //clear flag

   OFFSET_CH0_DIFF += ADCA.CH0.RES;
   OFFSET_CH1_DIFF += ADCA.CH1.RES;
   OFFSET_CH2_DIFF += ADCA.CH2.RES;
}

   OFFSET_CH0_DIFF = OFFSET_CH0_TOTAL>>6;
   OFFSET_CH1_DIFF = OFFSET_CH1_TOTAL>>6;
   OFFSET_CH2_DIFF = OFFSET_CH2_TOTAL>>6;

  ADCA.CH0.MUXCTRL = 0b00000011; // Differential POS=A0 and NEG=A7,
Battery Voltage
  ADCA.CH1.MUXCTRL = 0b00010010; // Differential POS=A2 and NEG=A6,
Battery Current
  ADCA.CH2.MUXCTRL = 0b00100001; // Differential POS=A4 and NEG=A5,
MOSFET Current

// Globaly enable interrupts
#asm("sei")


//Battery Voltage Control
KpV = -200;
```

```
refV = STOP_CHARGE_VOLT;

while (1)
    {
        if(stepready){ //EVERY 4 ms
            stepready=0;

            //READ NEW OVERSAMPLED ANALOG VALUES
            newAdcVal[0] = ADCvalues[0]; //Raw Batt Voltage
            newAdcVal[1] = ADCvalues[1]; //Raw Batt Current
            newAdcVal[2] = ADCvalues[2]; //Raw MOSFET Current

            //PLACE INTO RING BUFFER LAST 64 SAMPLES
            if(initSamples<64){ //Grabs first 64 samples to start off average filter, this
all occurs during INIT MODE so battVoltageFilt can be wrong during this time
                for (i=0; i<3; i++){
                    movAvgValues[i][initSamples] = newAdcVal[i];
                    movAvgTotal[i] = movAvgTotal[i] +
movAvgValues[i][initSamples]; //Eventually will hold total of first 128 samples
                }
                initSamples++;
            }else{
                for (i=0; i<3; i++){
                    movAvgTotal[i] = movAvgTotal[i] -
movAvgValues[i][sampleIdxOldest]; //Subtracts out oldest sample
                    movAvgValues[i][sampleIdxOldest] = newAdcVal[i]; //Replace
oldest sample value with a new value
                    movAvgTotal[i] =
movAvgTotal[i]+movAvgValues[i][sampleIdxOldest]; //Add in newest value,
movAvgTotal is now last 128 values, increment oldest index
                }
                sampleIdxOldest++;
                sampleIdxOldest&=0x3F; //Wraps around Oldest Index value in case it
"overflows" out of range of movAvgValues[] array
            }

            //CALCULATE AVERAGE WITH BITSHIFT
            battVoltRaw = movAvgTotal[0]>>6;
            battCurrRaw = movAvgTotal[1]>>6;
            mosCurrRaw = movAvgTotal[2]>>6;
            tempRaw = ADCvalues[3];

            //CORRECT READINGS WITH CALIBRATED VALUES
```

```
        offset = (signed int)(ADC_OFFSET_M[slaveAddress-1]*mosCurrRaw +
ADC_OFFSET_B[slaveAddress-1]);
        mosCurrRaw = mosCurrRaw+offset;

        offset = (signed int)(ADC_OFFSET_M[slaveAddress-1]*battVoltRaw +
ADC_OFFSET_B[slaveAddress-1]);
        battVoltRaw = battVoltRaw+offset;

        //CONVERT RAW ADC VALUES TO SCIENTIFIC NOTATION
        battVolt =  battVoltRaw*0.0025;
        battCurr = battCurrRaw*0.020833333; //6 mOhm, gain 8
        mosCurr = mosCurrRaw*.001; //1 ohm, gain 1
        temp = -1481.96 + sqrt(2196200 + (1863.9-tempRaw)*257.732);
        tempRaw = (unsigned int)(temp*100);
        socRaw = (unsigned
int)((float)4095.0*(float)((float)AmpSec/(float)MAX_AMP_SEC));

        //SATURATE MOSFET CURRENT VALUE FOR NOISE REASONS
        if (mosCurr<0) mosCurr=0;

        //CREATE PACKET FOR DATA TO BE SENT TO MASTER
        battData[0] = (unsigned char)(battCurrRaw & 0x00FF);
        battData[1] = (unsigned char)(battCurrRaw >> 8);
        battData[2] = (unsigned char)(battVoltRaw & 0x00FF);
        battData[3] = (unsigned char)(battVoltRaw >> 8);
        battData[4] = (unsigned char)(mosCurrRaw & 0x00FF);
        battData[5] = (unsigned char)(mosCurrRaw >> 8);
        battData[6] = (unsigned char)(tempRaw & 0x00FF);
        battData[7] = (unsigned char)(tempRaw >> 8);
        battData[8] = (unsigned char)(socRaw & 0x00FF);
        battData[9] = (unsigned char)(((socRaw >> 8)&0x0F) | byPassOn<<7 |
byPassMax<<6 | overDischarge<<5 | overTemp<<4);

        //UPDATE SINGLE BYPASS STATUS LED TO REFLECT BYPASS
STATE
        if(mosCurr>1.90){
            ledColor=RED;
            byPassMax=1;
        }else if(mosCurr>0.05){
            ledColor=AMBER;
            byPassOn=1;
        }else{
            ledColor=GREEN;
            byPassOn=0;
```

```
            byPassMax=0;
        }

        //BATTERY OVERDISCHARGE STATUS
        if(battVolt<OVERDISCHARGE_VOLT){
            overDischarge=1;
        }else{
            overDischarge=0;
        }

        //BYPASS ACTIVATION LOGIC AND CONTROL LOOP
        voltCntrlCnt++;
         if(voltCntrlCnt>75){
            voltCntrlCnt=0;
            error = (refV-battVolt);
            if (error>=0){ //if batteryVoltage <3.8 means error is positive
               if((duty+error*KpV)<=0){  //prevent controller from causing duty
from rolling over
                  duty=0;
               }else{
                  duty=duty+KpV*error;
               }
            }else{
               duty=duty+KpV*error;
            }
         }


        //SATURATE MAX DUTY CYCLE
        if(duty>DUTY_CYC_MAX) duty=DUTY_CYC_MAX;

        //Ensure mosfet stays off if battery is not charging and battery voltage is
less than max
        if(battCurr<=0.05 && battVolt<STOP_CHARGE_VOLT) {
            if(dutyOffCnt<=MAX_DUTY_OFF_CNT) dutyOffCnt++;

            if(dutyOffCnt>=MAX_DUTY_OFF_CNT){
               //duty=0;
               //dutyOffCnt=0;
               TCC1.CCA=0;
            }
        }else{
            TCC1.CCA=duty;
        }
```

```
    }

    switch(ledVal){
        case 1:
            PORTD.OUT = 0;
            PORTE.OUT = 0;
            PORTD.OUTSET = ledColor; //0b00000010==2
            break;
        case 2:
            PORTD.OUT = 0;
            PORTE.OUT = 0;
            PORTD.OUTSET = ledColor<<2; //0b00001000==8
            break;
        case 3:
            PORTD.OUT = 0;
            PORTE.OUT = 0;
            PORTD.OUTSET = ledColor<<4; //0b00100000==32
            break;
        case 4:
            PORTD.OUT = 0;
            PORTE.OUT = 0;
            PORTD.OUTSET = ledColor<<6; //0b10000000==128
            break;
        case 5:
            PORTD.OUT = 0;
            PORTE.OUT = 0;
            PORTE.OUTSET = ledColor; //0b00000010==2
            break;
        case 6:
            PORTD.OUT = 0;
            PORTE.OUT = 0;
            PORTE.OUTSET = ledColor<<2; //0b00001000==8
            break;
        case LOW_SOC:
            PORTD.OUTTGL = 0b01010101;
            PORTE.OUTTGL = 0b00000101;
            break;
```

APPENDIX C

ICMU/BPMU HYBRID CODE

```
/*****************************************************
This program was produced by the
CodeWizardAVR V2.04.8 Standard
Automatic Program Generator
 Copyright 1998-2010 Pavel Haiduc, HP InfoTech s.r.l.
http://www.hpinfotech.com

Project : Battery Board Slave
Version : .9
Date    : 10/15/2010
Author  :
Company : University of Akron
Comments:


Chip type               : ATxmega16A4
Program type            : Application
AVR Core Clock frequency: 32.000000 MHz
Memory model            : Small
Data Stack size         : 512
*****************************************************/

#include <io.h>
#include <math.h>
#include <stdio.h>
#include <avr_compiler.h>

// Declare your global variables here

#define MAX_AMP_SEC 216000.00
#define STOP_CHARGE_VOLT 3.8
#define OVERDISCHARGE_VOLT 2.5
//Equivalent to a voltage of 3.5V
#define TSamp_TCD1 0.0039996 //Current Integrating time in seconds

#define DUTY_CYC_MAX 1330 //Limits Max Bypass Current to ~2 Amps at
3.8 Volts
#define WAIT_BEFORE_CHANGE_CNT 50000 //2500 COUNTS per
second...Roughly 20 seconds
#define MAX_TURNOFF_CNT 5000 //2500 COUNTS per second seconds
```

```
#define MAX_BYPASS_CURRENT 1.9

//Count to wait until MOSFET locks on after battVolt>3.8V
mosCurrent>1.95A and battCurr<1A
#define MAX_LOCK_MOS_CURRENT_CNT 700

//Count to wait until Duty Cycle Shuts off after seeing current out
of battery
#define MAX_DUTY_OFF_CNT 700

//Address for all slave boards to respond to
#define allCallAddress 100

#define READ 1
#define WRITE 0

#define MAX_BATT_CNT 10

unsigned int dutyOffCnt=0;

//LED Colors
#define GREEN 0b10
#define RED 0b01
#define AMBER 0b11

#define LOW_SOC 70 //Picked a value then the largest ledValue

//#define IS_CHARGING_AT_2_MSG 0xF0
//#define IS_NOT_CHARGING_AT_2_MSG 0xFA

unsigned char ledColor=GREEN;

unsigned char underVoltCond=0;

unsigned int waitToChangeCnt=65531; //No waiting initially
unsigned char sendDesiredCurrent=1;
//unsigned char lockMosCurr = 0;

unsigned char allBypassOn=1;
unsigned int turnOffChargerCnt=0;

unsigned char overVoltageShutoff=0;
unsigned int turnOffChargerCnt2=0;

unsigned int overTemperatureShutoff=0;
unsigned int turnOffChargerCnt3=0;

unsigned int underVoltCondCnt=0;

float temp[14];
float battVolt[14];
float battCurr[14];
float mosCurr[14];
float soc[14];
```

```
unsigned char byPassOn[14];
unsigned char byPassMax[14];
unsigned char overDischarge[14];
unsigned char overTemp[14];

//unsigned char chargeStatus;
unsigned char twiLock = 0;

float packCurr=0;
float packSoc=100;
float packVolt=0;

//float prevBattCurr;
float AmpSec=MAX_AMP_SEC;

unsigned int voltCntrlCnt=0;
unsigned int decreaseCurrentCnt=0;

unsigned char currentPkt[6];

unsigned char battBypassId=0;
unsigned char prevBattBypassId=0;

eeprom unsigned char desiredCurrentIntEE=10;
eeprom unsigned char desiredCurrentDecEE=0;

unsigned char desiredCurrentInt=0;
unsigned char desiredCurrentDec=0;

signed long OFFSET_CH0_TOTAL=0;
signed long OFFSET_CH1_TOTAL=0;
signed long OFFSET_CH2_TOTAL=0;

signed int OFFSET_CH0_DIFF=0;
signed int OFFSET_CH1_DIFF=0;
signed int OFFSET_CH2_DIFF=0;

signed long ADCtemp[4];
signed int ADCvalues[4];
char ADCsamplecount  = 0; // we're supersampling
signed int ADCbuffer;
char stepready = 0;

unsigned char increment=1;
unsigned char dataCnt=0;
unsigned char dataOutTWIC=0;

unsigned char slaveAddress=0;
unsigned char currAddress=0;

unsigned char newData = 0;
unsigned char dataRdy = 0;

unsigned char battIdx=0;
unsigned char dataIdx=0;
```

```
unsigned char battData[14][10];

//ADC OFFSET for each of the 10 boards OFFSET=M*RAWADC+B calibrated
manually
float const ADC_OFFSET_M[10]={-0.011107,-0.013504,-0.015376,-
0.017009,-0.015646,-0.013056,-0.015416,-0.015210,-0.013949,-
0.012555};
float const ADC_OFFSET_B[10]={ 6.366141, 6.417454, 6.556141,
7.638119, 6.499410, 5.792097, 5.235612, 3.654257, 5.374430,
5.624966};

signed int offset=0;

signed int bound(signed int value, signed int min, signed int max){
    if(value > max)
        return max;
    if(value < min)
        return min;
    return value;
}


#pragma warn-
char nvm_cmd_read( char *nvm_cmd_addr, char index ){
    #asm
        LDD  R30,Y+0     ; Z = index
        LDI  R31,0
         LDD  R26,Y+1     ; X = &NVM.CMD
         LDD  R27,Y+2
        LDI  R25,2       ; NVM.CMD = NVM_CMD_READ_CALIB_ROW_gc
        ST   X,R25
        LPM              ; read the data in R0
                          ; Clean up NVM Command register. */
        LDI  R25,0       ; NVM.CMD = NVM_CMD_NO_OPERATION_gc
        ST   X,R25
        MOV  R30,R0      ; return result
    #endasm
}
#pragma warn+

char SP_ReadCalibrationByte( char index ){
    return nvm_cmd_read(&NVM.CMD,index);
}


// System Clocks initialization
void system_clocks_init(void)
{
unsigned char n,s;

// Optimize for speed
#pragma optsize-
// Save interrupts enabled/disabled state
s=SREG;
// Disable interrupts
```

```
#asm("cli")

// Internal 32 kHz RC oscillator initialization
// Enable the internal 32 kHz RC oscillator
OSC.CTRL|=OSC_RC32KEN_bm;
// Wait for the internal 32 kHz RC oscillator to stabilize
while ((OSC.STATUS & OSC_RC32KRDY_bm)==0);

// Internal 32 MHz RC oscillator initialization
// Enable the internal 32 MHz RC oscillator
OSC.CTRL|=OSC_RC32MEN_bm;

// System Clock prescaler A division factor: 1
// System Clock prescalers B & C division factors: B:1, C:1
// ClkPer4: 32000.000 kHz
// ClkPer2: 32000.000 kHz
// ClkPer:  32000.000 kHz
// ClkCPU:  32000.000 kHz
n=(CLK.PSCTRL & (~(CLK_PSADIV_gm | CLK_PSBCDIV1_bm |
CLK_PSBCDIV0_bm))) |
    CLK_PSADIV_1_gc | CLK_PSBCDIV_1_1_gc;
CCP=CCP_IOREG_gc;
CLK.PSCTRL=n;

// Internal 32 MHz RC osc. calibration reference clock source:
32.768 kHz Internal Osc.
OSC.DFLLCTRL&= ~(OSC_RC32MCREF_bm | OSC_RC2MCREF_bm);
// Enable the autocalibration of the internal 32 MHz RC oscillator
DFLLRC32M.CTRL|=DFLL_ENABLE_bm;

// Wait for the internal 32 MHz RC oscillator to stabilize
while ((OSC.STATUS & OSC_RC32MRDY_bm)==0);

// Select the system clock source: 32 MHz Internal RC Osc.
n=(CLK.CTRL & (~CLK_SCLKSEL_gm)) | CLK_SCLKSEL_RC32M_gc;
CCP=CCP_IOREG_gc;
CLK.CTRL=n;

// Disable the unused oscillators: 2 MHz, external clock/crystal
oscillator, PLL
OSC.CTRL&= ~(OSC_RC2MEN_bm | OSC_XOSCEN_bm | OSC_PLLEN_bm);

// Peripheral Clock output: Disabled
PORTCFG.CLKEVOUT=(PORTCFG.CLKEVOUT & (~PORTCFG_CLKOUT_gm)) |
PORTCFG_CLKOUT_OFF_gc;

// Restore interrupts enabled/disabled state
SREG=s;

}

// Watchdog Timer initialization
void watchdog_init(void)
{
unsigned char s,n;
```

```
// Optimize for speed
#pragma optsize-
// Save interrupts enabled/disabled state
s=SREG;
// Disable interrupts
#asm("cli")

// Watchdog Timer: On
n=WDT_ENABLE_bm | WDT_CEN_bm | WDT_PER_256CLK_gc;
CCP=CCP_IOREG_gc;
WDT.CTRL=n;

// Watchdog window mode: Off
n=(WDT.WINCTRL & (~WDT_WEN_bm)) | WDT_WCEN_bm;
CCP=CCP_IOREG_gc;
WDT.WINCTRL=n;

// Restore interrupts enabled/disabled state
SREG=s;

}

// Event System initialization
void event_system_init(void)
{
// Event System Channel 0 source: Port D, Pin0
EVSYS.CH0MUX=EVSYS_CHMUX_PORTD_PIN0_gc;
// Event System Channel 1 source: Port D, Pin0
EVSYS.CH1MUX=EVSYS_CHMUX_OFF_gc;

// Event System Channel 2 source: None
EVSYS.CH2MUX=EVSYS_CHMUX_PORTD_PIN1_gc;
// Event System Channel 3 source: None
EVSYS.CH3MUX=EVSYS_CHMUX_OFF_gc;
// Event System Channel 4 source: None
EVSYS.CH4MUX=EVSYS_CHMUX_OFF_gc;
// Event System Channel 5 source: None
EVSYS.CH5MUX=EVSYS_CHMUX_OFF_gc;
// Event System Channel 6 source: None
EVSYS.CH6MUX=EVSYS_CHMUX_OFF_gc;
// Event System Channel 7 source: None
EVSYS.CH7MUX=EVSYS_CHMUX_OFF_gc;

// Event System Channel 0 Digital Filter Coefficient: 4 Samples
EVSYS.CH0CTRL=0b00001011;
// Event System Channel 1 Digital Filter Coefficient: 1 Sample
EVSYS.CH1CTRL=EVSYS_DIGFILT_1SAMPLE_gc;
// Event System Channel 2 Digital Filter Coefficient: 1 Sample
EVSYS.CH2CTRL=0b00001011;
// Event System Channel 3 Digital Filter Coefficient: 1 Sample
EVSYS.CH3CTRL=EVSYS_DIGFILT_1SAMPLE_gc;
// Event System Channel 4 Digital Filter Coefficient: 1 Sample
EVSYS.CH4CTRL=(EVSYS.CH4CTRL & (~(EVSYS_QDIRM_gm | EVSYS_QDIEN_bm |
EVSYS_QDEN_bm | EVSYS_DIGFILT_gm))) |
```

```
    EVSYS_DIGFILT_1SAMPLE_gc;
// Event System Channel 5 Digital Filter Coefficient: 1 Sample
EVSYS.CH5CTRL=EVSYS_DIGFILT_1SAMPLE_gc;
// Event System Channel 6 Digital Filter Coefficient: 1 Sample
EVSYS.CH6CTRL=EVSYS_DIGFILT_1SAMPLE_gc;
// Event System Channel 7 Digital Filter Coefficient: 1 Sample
EVSYS.CH7CTRL=EVSYS_DIGFILT_1SAMPLE_gc;

// Event System Channel 0 output: Disabled
// Note: the correct direction for the Event System Channel 0
output
// is configured in the ports_init function
PORTCFG.CLKEVOUT=(PORTCFG.CLKEVOUT & (~PORTCFG_EVOUT_gm)) |
PORTCFG_EVOUT_OFF_gc;
//PORTCFG.CLKEVOUT=0b00000001;

}

// Ports initialization
void ports_init(void)
{
// PORTA initialization
// OUT register
PORTA.OUT=0x00;

// Bit0: Input
// Bit1: Input
// Bit2: Input
// Bit3: Input
// Bit4: Input
// Bit5: Input
// Bit6: Input
// Bit7: Input
PORTA.DIR=0x00;

// Bit0 Output/Pull configuration: Totempole/No
// Bit0 Input/Sense configuration: Input buffer disabled
// Bit0 inverted: Off
// Bit0 slew rate limitation: Off
PORTA.PIN0CTRL=PORT_OPC_TOTEM_gc | PORT_ISC_INPUT_DISABLE_gc;
// Bit1 Output/Pull configuration: Totempole/No
// Bit1 Input/Sense configuration: Input buffer disabled
// Bit1 inverted: Off
// Bit1 slew rate limitation: Off
PORTA.PIN1CTRL=PORT_OPC_TOTEM_gc | PORT_ISC_INPUT_DISABLE_gc;
// Bit2 Output/Pull configuration: Totempole/No
// Bit2 Input/Sense configuration: Input buffer disabled
// Bit2 inverted: Off
// Bit2 slew rate limitation: Off
PORTA.PIN2CTRL=PORT_OPC_TOTEM_gc | PORT_ISC_INPUT_DISABLE_gc;
// Bit3 Output/Pull configuration: Totempole/No
// Bit3 Input/Sense configuration: Input buffer disabled
// Bit3 inverted: Off
// Bit3 slew rate limitation: Off
PORTA.PIN3CTRL=PORT_OPC_TOTEM_gc | PORT_ISC_INPUT_DISABLE_gc;
```

```
// Bit4 Output/Pull configuration: Totempole/No
// Bit4 Input/Sense configuration: Input buffer disabled
// Bit4 inverted: Off
// Bit4 slew rate limitation: Off
PORTA.PIN4CTRL=PORT_OPC_TOTEM_gc | PORT_ISC_INPUT_DISABLE_gc;
// Bit5 Output/Pull configuration: Totempole/No
// Bit5 Input/Sense configuration: Input buffer disabled
// Bit5 inverted: Off
// Bit5 slew rate limitation: Off
PORTA.PIN5CTRL=PORT_OPC_TOTEM_gc | PORT_ISC_INPUT_DISABLE_gc;
// Bit6 Output/Pull configuration: Totempole/No
// Bit6 Input/Sense configuration: Input buffer disabled
// Bit6 inverted: Off
// Bit6 slew rate limitation: Off
PORTA.PIN6CTRL=PORT_OPC_TOTEM_gc | PORT_ISC_INPUT_DISABLE_gc;
// Bit7 Output/Pull configuration: Totempole/No
// Bit7 Input/Sense configuration: Input buffer disabled
// Bit7 inverted: Off
// Bit7 slew rate limitation: Off
PORTA.PIN7CTRL=PORT_OPC_TOTEM_gc | PORT_ISC_INPUT_DISABLE_gc;
// Interrupt 0 level: Disabled
// Interrupt 1 level: Disabled
PORTA.INTCTRL=(PORTA.INTCTRL & (~(PORT_INT1LVL_gm |
PORT_INT0LVL_gm))) |
    PORT_INT1LVL_OFF_gc | PORT_INT0LVL_OFF_gc;
// Bit0 pin change interrupt 0: Off
// Bit1 pin change interrupt 0: Off
// Bit2 pin change interrupt 0: Off
// Bit3 pin change interrupt 0: Off
// Bit4 pin change interrupt 0: Off
// Bit5 pin change interrupt 0: Off
// Bit6 pin change interrupt 0: Off
// Bit7 pin change interrupt 0: Off
PORTA.INT0MASK=0x00;
// Bit0 pin change interrupt 1: Off
// Bit1 pin change interrupt 1: Off
// Bit2 pin change interrupt 1: Off
// Bit3 pin change interrupt 1: Off
// Bit4 pin change interrupt 1: Off
// Bit5 pin change interrupt 1: Off
// Bit6 pin change interrupt 1: Off
// Bit7 pin change interrupt 1: Off
PORTA.INT1MASK=0x00;

// PORTB initialization
// OUT register
PORTB.OUT=0x00;
// Bit0: Input
// Bit1: Input
// Bit2: Input
// Bit3: Input
PORTB.DIR=0x00;
// Bit0 Output/Pull configuration: Totempole/No
// Bit0 Input/Sense configuration: Input buffer disabled
// Bit0 inverted: Off
```

```
// Bit0 slew rate limitation: Off
PORTB.PIN0CTRL=PORT_OPC_TOTEM_gc | PORT_ISC_INPUT_DISABLE_gc;
// Bit1 Output/Pull configuration: Totempole/No
// Bit1 Input/Sense configuration: Input buffer disabled
// Bit1 inverted: Off
// Bit1 slew rate limitation: Off
PORTB.PIN1CTRL=PORT_OPC_TOTEM_gc | PORT_ISC_INPUT_DISABLE_gc;
// Bit2 Output/Pull configuration: Totempole/No
// Bit2 Input/Sense configuration: Input buffer disabled
// Bit2 inverted: Off
// Bit2 slew rate limitation: Off
PORTB.PIN2CTRL=PORT_OPC_TOTEM_gc | PORT_ISC_INPUT_DISABLE_gc;
// Bit3 Output/Pull configuration: Totempole/No
// Bit3 Input/Sense configuration: Input buffer disabled
// Bit3 inverted: Off
// Bit3 slew rate limitation: Off
PORTB.PIN3CTRL=PORT_OPC_TOTEM_gc | PORT_ISC_INPUT_DISABLE_gc;
// Interrupt 0 level: Disabled
// Interrupt 1 level: Disabled
PORTB.INTCTRL=(PORTB.INTCTRL & (~(PORT_INT1LVL_gm |
PORT_INT0LVL_gm))) |
    PORT_INT1LVL_OFF_gc | PORT_INT0LVL_OFF_gc;
// Bit0 pin change interrupt 0: Off
// Bit1 pin change interrupt 0: Off
// Bit2 pin change interrupt 0: Off
// Bit3 pin change interrupt 0: Off
PORTB.INT0MASK=0x00;
// Bit0 pin change interrupt 1: Off
// Bit1 pin change interrupt 1: Off
// Bit2 pin change interrupt 1: Off
// Bit3 pin change interrupt 1: Off
PORTB.INT1MASK=0x00;


// Bit0: Output
// Bit1: Output
// Bit2: Input
// Bit3: Input
// Bit4: Output
// Bit5: Input
// Bit6: Input
// Bit7: Output
PORTC.DIR=0x93;

// Bit0 Output/Pull configuration: WIRED-AND (on input)
// Bit0 Input/Sense configuration: Sense both edges
// Bit0 inverted: Off
// Bit0 slew rate limitation: Off
PORTC.PIN0CTRL=PORT_OPC_WIREDAND_gc | PORT_ISC_BOTHEDGES_gc;
// Bit1 Output/Pull configuration: WIRED-AND (on input)
// Bit1 Input/Sense configuration: Sense both edges
// Bit1 inverted: Off
// Bit1 slew rate limitation: Off
PORTC.PIN1CTRL=PORT_OPC_WIREDAND_gc | PORT_ISC_BOTHEDGES_gc;
PORTC.PIN2CTRL=PORT_OPC_TOTEM_gc | PORT_ISC_BOTHEDGES_gc;
```

```
// Bit3 Output/Pull configuration: Totempole/No
// Bit3 Input/Sense configuration: Sense both edges
// Bit3 inverted: Off
// Bit3 slew rate limitation: Off
PORTC.PIN3CTRL=PORT_OPC_TOTEM_gc | PORT_ISC_BOTHEDGES_gc;
// Bit4 Output/Pull configuration: Totempole/Pull-up (on input)
// Bit4 Input/Sense configuration: Sense both edges
// Bit4 inverted: Off
// Bit4 slew rate limitation: Off
PORTC.PIN4CTRL=PORT_OPC_PULLDOWN_gc | PORT_ISC_BOTHEDGES_gc;
// Bit5 Output/Pull configuration: Totempole/Pull-up (on input)
// Bit5 Input/Sense configuration: Sense both edges
// Bit5 inverted: Off
// Bit5 slew rate limitation: Off
PORTC.PIN5CTRL=PORT_OPC_PULLUP_gc | PORT_ISC_BOTHEDGES_gc;
// Bit6 Output/Pull configuration: Totempole/Pull-up (on input)
// Bit6 Input/Sense configuration: Sense both edges
// Bit6 inverted: Off
// Bit6 slew rate limitation: Off
PORTC.PIN6CTRL=PORT_OPC_PULLUP_gc | PORT_ISC_BOTHEDGES_gc;
// Bit7 Output/Pull configuration: Totempole/Pull-up (on input)
// Bit7 Input/Sense configuration: Sense both edges
// Bit7 inverted: Off
// Bit7 slew rate limitation: Off
PORTC.PIN7CTRL=PORT_OPC_TOTEM_gc | PORT_ISC_BOTHEDGES_gc;
// Interrupt 0 level: Disabled
// Interrupt 1 level: Disabled
PORTC.INTCTRL=(PORTC.INTCTRL & (~(PORT_INT1LVL_gm |
PORT_INT0LVL_gm))) |
    PORT_INT1LVL_OFF_gc | PORT_INT0LVL_OFF_gc;
// Bit0 pin change interrupt 0: Off
// Bit1 pin change interrupt 0: Off
// Bit2 pin change interrupt 0: Off
// Bit3 pin change interrupt 0: Off
// Bit4 pin change interrupt 0: Off
// Bit5 pin change interrupt 0: Off
// Bit6 pin change interrupt 0: Off
// Bit7 pin change interrupt 0: Off
PORTC.INT0MASK=0x00;
// Bit0 pin change interrupt 1: Off
// Bit1 pin change interrupt 1: Off
// Bit2 pin change interrupt 1: Off
// Bit3 pin change interrupt 1: Off
// Bit4 pin change interrupt 1: Off
// Bit5 pin change interrupt 1: Off
// Bit6 pin change interrupt 1: Off
// Bit7 pin change interrupt 1: Off
PORTC.INT1MASK=0x00;

// PORTD initialization
// OUT register
PORTD.OUT=0x00;

// Bit0: Output
// Bit1: Output
```

```
// Bit2: Output
// Bit3: Output
// Bit4: Output
// Bit5: Output
// Bit6: Output
// Bit7: Output
PORTD.DIR=0xFF;

// Bit0 Output/Pull configuration: Totempole/Pull-up (on input)
// Bit0 Input/Sense configuration: Sense low level
// Bit0 inverted: Off
// Bit0 slew rate limitation: Off
PORTD.PIN0CTRL=PORT_OPC_TOTEM_gc | PORT_ISC_LEVEL_gc;

// Bit1 Output/Pull configuration: Totempole/Pull-up (on input)
// Bit1 Input/Sense configuration: Sense low level
// Bit1 inverted: Off
// Bit1 slew rate limitation: Off
PORTD.PIN1CTRL=PORT_OPC_TOTEM_gc | PORT_ISC_LEVEL_gc;
// Bit2 Output/Pull configuration: Totempole/Pull-up (on input)
// Bit2 Input/Sense configuration: Sense both edges
// Bit2 inverted: Off
// Bit2 slew rate limitation: Off
PORTD.PIN2CTRL=PORT_OPC_TOTEM_gc | PORT_ISC_BOTHEDGES_gc;
// Bit3 Output/Pull configuration: Totempole/Pull-up (on input)
// Bit3 Input/Sense configuration: Sense both edges
// Bit3 inverted: Off
// Bit3 slew rate limitation: Off
PORTD.PIN3CTRL=PORT_OPC_TOTEM_gc | PORT_ISC_BOTHEDGES_gc;
// Bit4 Output/Pull configuration: Totempole/Pull-up (on input)
// Bit4 Input/Sense configuration: Sense both edges
// Bit4 inverted: Off
// Bit4 slew rate limitation: Off
PORTD.PIN4CTRL=PORT_OPC_TOTEM_gc | PORT_ISC_BOTHEDGES_gc;
// Bit5 Output/Pull configuration: Totempole/Pull-up (on input)
// Bit5 Input/Sense configuration: Sense both edges
// Bit5 inverted: Off
// Bit5 slew rate limitation: Off
PORTD.PIN5CTRL=PORT_OPC_TOTEM_gc | PORT_ISC_BOTHEDGES_gc;
// Bit6 Output/Pull configuration: Totempole/No
// Bit6 Input/Sense configuration: Sense both edges
// Bit6 inverted: Off
// Bit6 slew rate limitation: Off
PORTD.PIN6CTRL=PORT_OPC_TOTEM_gc | PORT_ISC_BOTHEDGES_gc;
// Bit7 Output/Pull configuration: Totempole/No
// Bit7 Input/Sense configuration: Sense both edges
// Bit7 inverted: Off
// Bit7 slew rate limitation: Off
PORTD.PIN7CTRL=PORT_OPC_TOTEM_gc | PORT_ISC_BOTHEDGES_gc;
// Interrupt 0 level: Disabled
// Interrupt 1 level: Disabled

PORTD.INTCTRL=(PORTD.INTCTRL & (~(PORT_INT1LVL_gm |
PORT_INT0LVL_gm))) |
    PORT_INT1LVL_OFF_gc | PORT_INT0LVL_OFF_gc;
```

```
// Bit0 pin change interrupt 0: Off
// Bit1 pin change interrupt 0: Off
// Bit2 pin change interrupt 0: Off
// Bit3 pin change interrupt 0: Off
// Bit4 pin change interrupt 0: Off
// Bit5 pin change interrupt 0: Off
// Bit6 pin change interrupt 0: Off
// Bit7 pin change interrupt 0: Off

PORTD.INT0MASK=0x00;
// Bit0 pin change interrupt 1: Off
// Bit1 pin change interrupt 1: Off
// Bit2 pin change interrupt 1: Off
// Bit3 pin change interrupt 1: Off
// Bit4 pin change interrupt 1: Off
// Bit5 pin change interrupt 1: Off
// Bit6 pin change interrupt 1: Off
// Bit7 pin change interrupt 1: Off
PORTD.INT1MASK=0x00;

// PORTE initialization
// OUT register
PORTE.OUT=0x00; //Initial Output Value

// Bit0: Output
// Bit1: Output
// Bit2: Input  //Charger Present Detection, Internal pullup,
Yellow Wire
// Bit3: Output //Serial Output PE3
PORTE.DIR=0x0B;

// Bit0 Output/Pull configuration: Totempole/No
// Bit0 Input/Sense configuration: Sense both edges
// Bit0 inverted: Off
// Bit0 slew rate limitation: Off
PORTE.PIN0CTRL=PORT_OPC_TOTEM_gc | PORT_ISC_BOTHEDGES_gc;

// Bit1 Output/Pull configuration: Totempole/No
// Bit1 Input/Sense configuration: Sense both edges
// Bit1 inverted: On
// Bit1 slew rate limitation: Off
PORTE.PIN1CTRL=PORT_OPC_TOTEM_gc | PORT_ISC_BOTHEDGES_gc;

// Bit2 Output/Pull configuration: Pull-up (on input)
// Bit2 Input/Sense configuration: Sense both edges
// Bit2 inverted: Off
// Bit2 slew rate limitation: Off
PORTE.PIN2CTRL=PORT_OPC_PULLUP_gc | PORT_ISC_BOTHEDGES_gc;

// Bit3 Output/Pull configuration: Totempole/Pull-up (on input)
// Bit3 Input/Sense configuration: Sense both edges
// Bit3 inverted: Off
// Bit3 slew rate limitation: Off
PORTE.PIN3CTRL=PORT_OPC_TOTEM_gc | PORT_ISC_BOTHEDGES_gc;
// Interrupt 0 level: Disabled
```

```
// Interrupt 1 level: Disabled

PORTE.INTCTRL=(PORTE.INTCTRL & (~(PORT_INT1LVL_gm |
PORT_INT0LVL_gm))) |
    PORT_INT1LVL_OFF_gc | PORT_INT0LVL_OFF_gc;
// Bit0 pin change interrupt 0: Off
// Bit1 pin change interrupt 0: Off
// Bit2 pin change interrupt 0: Off
// Bit3 pin change interrupt 0: Off
PORTE.INT0MASK=0x00;
// Bit0 pin change interrupt 1: Off
// Bit1 pin change interrupt 1: Off
// Bit2 pin change interrupt 1: Off
// Bit3 pin change interrupt 1: Off
PORTE.INT1MASK=0x00;

// PORTR initialization
// OUT register
PORTR.OUT=0x00;
// Bit0: Input
// Bit1: Input
PORTR.DIR=0x00;
// Bit0 Output/Pull configuration: Totempole/No
// Bit0 Input/Sense configuration: Sense both edges
// Bit0 inverted: Off
// Bit0 slew rate limitation: Off
PORTR.PIN0CTRL=PORT_OPC_TOTEM_gc | PORT_ISC_BOTHEDGES_gc;
// Bit1 Output/Pull configuration: Totempole/No
// Bit1 Input/Sense configuration: Sense both edges
// Bit1 inverted: Off
// Bit1 slew rate limitation: Off
PORTR.PIN1CTRL=PORT_OPC_TOTEM_gc | PORT_ISC_BOTHEDGES_gc;
// Interrupt 0 level: Disabled
// Interrupt 1 level: Disabled
PORTR.INTCTRL=(PORTR.INTCTRL & (~(PORT_INT1LVL_gm |
PORT_INT0LVL_gm))) |
    PORT_INT1LVL_OFF_gc | PORT_INT0LVL_OFF_gc;
// Bit0 pin change interrupt 0: Off
// Bit1 pin change interrupt 0: Off
PORTR.INT0MASK=0x00;
// Bit0 pin change interrupt 1: Off
// Bit1 pin change interrupt 1: Off
PORTR.INT1MASK=0x00;
}

// Disable a Timer/Counter type 0
void tc0_disable(TC0_t *ptc)
{
// Timer/Counter off
ptc->CTRLA=(ptc->CTRLA & (~TC0_CLKSEL_gm)) | TC_CLKSEL_OFF_gc;
// Issue a reset command
ptc->CTRLFSET=TC_CMD_RESET_gc;
}

// Disable a Timer/Counter type 1
```

```
void tc1_disable(TC1_t *ptc)
{
// Timer/Counter off
ptc->CTRLA=(ptc->CTRLA & (~TC1_CLKSEL_gm)) | TC_CLKSEL_OFF_gc;
// Issue a reset command
ptc->CTRLFSET=TC_CMD_RESET_gc;
}



// Timer/counter TCC1 Overflow/Underflow interrupt service routine
interrupt [TCC0_OVF_vect] void tcc0_overflow_isr(void)
{
// write your code here

}


// Timer/Counter TCD0 initialization
void tcd0_init(void)
{
unsigned char s;

// Note: the correct PORTD direction for the Compare Channels
outputs
// is configured in the ports_init function

// Save interrupts enabled/disabled state
s=SREG;
// Disable interrupts
#asm("cli")

// Disable and reset the timer/counter just to be sure
tc0_disable(&TCD0);
// Clock source: Peripheral Clock/1

TCD0.CTRLB=0b00000000;
TCD0.CTRLC=0b00000000;
TCD0.CTRLD=0b00000000;
TCD0.CTRLE=0b00000000;

// Overflow interrupt: Medium Level
// Error interrupt: Disabled
TCD0.INTCTRLA=(TCD0.INTCTRLA & (~(TC0_ERRINTLVL_gm |
TC0_OVFINTLVL_gm))) |
    TC_ERRINTLVL_OFF_gc | TC_OVFINTLVL_MED_gc;


// Compare/Capture channel A interrupt: Disabled
// Compare/Capture channel B interrupt: Disabled
// Compare/Capture channel C interrupt: Disabled
// Compare/Capture channel D interrupt: Disabled
TCD0.INTCTRLB=(TCD0.INTCTRLB & (~(TC0_CCDINTLVL_gm |
TC0_CCCINTLVL_gm | TC0_CCBINTLVL_gm | TC0_CCAINTLVL_gm))) |
```

```
    TC_CCDINTLVL_OFF_gc | TC_CCCINTLVL_OFF_gc | TC_CCBINTLVL_OFF_gc
| TC_CCAINTLVL_OFF_gc;

// High resolution extension: Off
HIRESD.CTRL&= ~HIRES_HREN0_bm;

// Clear the interrupt flags
TCD0.INTFLAGS=TCD0.INTFLAGS;

// Set counter register
TCD0.CNT=0x0000;
// Set period register
TCD0.PER=312;
// Set channel A Compare/Capture register
TCD0.CCA=0x0000;
// Set channel B Compare/Capture register
TCD0.CCB=0x0000;
// Set channel C Compare/Capture register
TCD0.CCC=0x0000;
// Set channel D Compare/Capture register
TCD0.CCD=0x0000;

// Restore interrupts enabled/disabled state
SREG=s;

TCD0.CTRLA=0b00000111;
}


// Timer/counter TCD0 Overflow/Underflow interrupt service routine
interrupt [TCD0_OVF_vect] void tcd0_overflow_isr(void)
{
    dataIdx=0;
    slaveAddress=battIdx+1;

    if(slaveAddress==1){
        newData=1;
    }else{
        twiLock=1;
        currAddress=slaveAddress<<1|READ;
        TWIC.MASTER.ADDR = currAddress; //slave address
    }

}



// Timer/Counter TCD1 initialization
void tcd1_init(void)
{
unsigned char s;

// Note: the correct PORTD direction for the Compare Channels
outputs
// is configured in the ports_init function
```

```
// Save interrupts enabled/disabled state
s=SREG;
// Disable interrupts
#asm("cli")

// Disable and reset the timer/counter just to be sure
tc1_disable(&TCD1);
// Clock source: Peripheral Clock/1
TCD1.CTRLA=(TCD1.CTRLA & (~TC1_CLKSEL_gm)) | TC_CLKSEL_DIV2_gc;

// Mode: Normal Operation, Overflow Int./Event on TOP
// Compare/Capture on channel A: Off
// Compare/Capture on channel B: Off
TCD1.CTRLB=(TCD1.CTRLB & (~(TC1_CCAEN_bm | TC1_CCBEN_bm |
TC1_WGMODE_gm))) |
    TC_WGMODE_NORMAL_gc;

// Capture event source: None
// Capture event action: None
TCD1.CTRLD=(TCD1.CTRLD & (~(TC1_EVACT_gm | TC1_EVSEL_gm))) |
    TC_EVACT_OFF_gc | TC_EVSEL_OFF_gc;

// Overflow interrupt: Low Level
// Error interrupt: Enabled
TCD1.INTCTRLA=(TCD1.INTCTRLA & (~(TC1_ERRINTLVL_gm ))) |
    TC_ERRINTLVL_OFF_gc | TC_OVFINTLVL_LO_gc | TC1_OVFINTLVL_gm;

// Compare/Capture channel A interrupt: Disabled
// Compare/Capture channel B interrupt: Disabled
TCD1.INTCTRLB=(TCD1.INTCTRLB & (~(TC1_CCBINTLVL_gm |
TC1_CCAINTLVL_gm))) |
    TC_CCBINTLVL_OFF_gc | TC_CCAINTLVL_OFF_gc;

// High resolution extension: Off
HIRESD.CTRL&= ~HIRES_HREN1_bm;

// Clear the interrupt flags
TCD1.INTFLAGS=TCD1.INTFLAGS;
// Set counter register
TCD1.CNT=0x0000;
// Set period register
TCD1.PER=64000;
// Set channel A Compare/Capture register
TCD1.CCA=0x0000;
// Set channel B Compare/Capture register
TCD1.CCB=0x0000;

// Restore interrupts enabled/disabled state
SREG=s;
}

// Timer/counter TCD1 Overflow/Underflow interrupt service routine
interrupt [TCD1_OVF_vect] void tcd1_overflow_isr(void){
    stepready = 1;
```

```
    if (battCurr[0]>0.1 || battCurr[0]<-0.35){
        AmpSec = AmpSec + battCurr[0]*TSamp_TCD1;
    }

    if(AmpSec<=0) AmpSec=0;
    else if(AmpSec>MAX_AMP_SEC) AmpSec=MAX_AMP_SEC;

}


// Timer/Counter TCE0 initialization
void tcc1_init(void)
{
unsigned char s;

// Note: the correct PORTC direction for the Compare Channels
outputs
// is configured in the ports_init function

// Save interrupts enabled/disabled state
s=SREG;
// Disable interrupts
#asm("cli")

// Disable and reset the timer/counter just to be sure
tc1_disable(&TCC1);
// Clock source: Peripheral Clock/1
TCC1.CTRLA=(TCC1.CTRLA & (~TC1_CLKSEL_gm)) | TC_CLKSEL_DIV1_gc;

// Mode: Dual Slope PWM Gen., Overflow Int./Event on TOP & BOTTOM
// Compare/Capture on channel A: Off
// Compare/Capture on channel B: On
// Compare/Capture on channel C: Off
// Compare/Capture on channel D: Off
//TCC0.CTRLB=(TCC0.CTRLB & (~(TC0_CCAEN_bm | TC0_CCBEN_bm |
TC0_CCCEN_bm | TC0_CCDEN_bm | TC0_WGMODE_gm))) |
//    TC0_CCBEN_bm | TC0_CCAEN_bm|
//    TC_WGMODE_DS_TB_gc;

TCC1.CTRLB = 0b00000011; //W
TCC1.CTRLB = TCC1.CTRLB | 0b00010000;


// Capture event source: None
// Capture event action: None
TCC1.CTRLD=(TCC0.CTRLD & (~(TC0_EVACT_gm | TC0_EVSEL_gm))) |
TC_EVACT_OFF_gc | TC_EVSEL_OFF_gc;

// Overflow interrupt: Medium Level
// Error interrupt: Disabled
// TCC0.INTCTRLA=(TCC0.INTCTRLA & (~(TC0_ERRINTLVL_gm |
TC0_OVFINTLVL_gm))) |
//    TC_ERRINTLVL_OFF_gc | TC_OVFINTLVL_MED_gc;
```

```
// Compare/Capture channel A interrupt: Disabled
// Compare/Capture channel B interrupt: Disabled

//TCC0.INTCTRLB=(TCC0.INTCTRLB & (~(TC0_CCDINTLVL_gm |
TC0_CCCINTLVL_gm | TC0_CCBINTLVL_gm | TC0_CCAINTLVL_gm))) |
//    TC_CCDINTLVL_OFF_gc | TC_CCCINTLVL_OFF_gc |
TC_CCBINTLVL_OFF_gc | TC_CCAINTLVL_OFF_gc;

// High resolution extension: Off
HIRESC.CTRL&= ~HIRES_HREN0_bm;

// Clear the interrupt flags
TCC1.INTFLAGS=0;

// Set counter value
TCC1.CNT=0x0000;

// Set period register
TCC1.PER=1600-1;

// Set channel A Compare/Capture register
TCC1.CCA=0;
// Set channel B Compare/Capture register
TCC1.CCB=0x0000;

// Restore interrupts enabled/disabled state
SREG=s;
}



// RTC initialization
void rtcxm_init(void)
{

// RTC 1.024kHz
RTC.CTRL=(RTC.CTRL & (~RTC_PRESCALER_gm)) | RTC_PRESCALER_DIV1_gc;

// RTC overflow interrupt: Off
// RTC compare interrupt: Off
RTC.INTCTRL=(RTC.INTCTRL & (~(RTC_OVFINTLVL_gm |
RTC_COMPINTLVL_gm))) |
    RTC_OVFINTLVL_OFF_gc | RTC_COMPINTLVL_OFF_gc;

RTC.CNT=0;

// 1.024kHz internal 32.768 RC oscilator, Enable RTC Clock
CLK.RTCCTRL = 0b0101;

}

//Code given from
    //http://blog.frankvh.com/2009/11/14/atmel-xmega-printf-howto/

// Init USART.  Transmit only (we're not receiving anything)
```

```
// We use USARTC1, transmit pin on PC7.
// Want 9600 baud. Have a 32 MHz clock. BSCALE = 0
// BSEL = ( 32000000 / (2^0 * 16*9600)) -1 = 103
// Fbaud = 32000000 / (2^0 * 16 * (12+1))  = 9615 bits/sec

void usartC1_init(void){
    // Set the TxD pin high - set PORTC DIR register bit 7 to 1
    PORTC.OUTSET = PIN7_bm;
    //PORTE.OUTSET = PIN3_bm;

    // Set the TxD pin as an output - set PORTC OUT register bit 7
to 1
    //PORTE.DIRSET = PIN3_bm;
    PORTC.DIRSET = PIN7_bm;

    // Set baud rate & frame format
    //USARTE0.BAUDCTRLB = 0;            // BSCALE = 0 as well,
19200
    //USARTE0.BAUDCTRLA = 0x67;


    USARTC1.BAUDCTRLB = 0b11010000;     // BSCALE = -3
    //USARTE0.BAUDCTRLB = 0b11010000;       // BSCALE = -3

    USARTC1.BAUDCTRLA = 135; //115,200
    //USARTE0.BAUDCTRLA = 63; //230,400
    //USARTE0.BAUDCTRLA = 27; //460800
    //USARTE0.BAUDCTRLA = 9; //921600
    //USARTC1.BAUDCTRLA = 9; //921600

    // Set mode of operation
    USARTC1.CTRLA = 0;              // no interrupts please
    USARTC1.CTRLC = 0x03;           // async, no parity, 8 bit
data, 1 stop bit
    //USARTE0.CTRLA = 0;                // no interrupts please
    //USARTE0.CTRLC = 0x03;         // async, no parity, 8 bit
data, 1 stop bit

    // Enable transmitter only
    USARTC1.CTRLB = 0b00001000;
    //USARTE0.CTRLB = 0b00001000;

}

void usartE0_init(void){
    // Set the TxD pin high - set PORTC DIR register bit 7 to 1
    PORTE.OUTSET = PIN3_bm;

    // Set the TxD pin as an output - set PORTC OUT register bit 7
to 1
    PORTE.DIRSET = PIN3_bm;

    // Set baud rate & frame format
    USARTE0.BAUDCTRLB = 0;          // BSCALE = 0 as well, 19200
    USARTE0.BAUDCTRLA = 0x67;
```

```
    // Set mode of operation
    USARTE0.CTRLA = 0;              // no interrupts please
    USARTE0.CTRLC = 0x03;           // async, no parity, 8 bit
data, 1 stop bit

    // Enable transmitter only
    USARTE0.CTRLB = 0b00001000;

}

void uartE0_putchar (char c)
{

    // Wait for the transmit buffer to be empty
    while ( !( USARTE0.STATUS & USART_DREIF_bm) );
    // Put our character into the transmit buffer
    USARTE0.DATA = c;
}



// Disable an USART
void usart_disable(USART_t *pu)
{
// Rx and Tx are off
pu->CTRLB=0;
// Ensure that all interrupts generated by the USART are off
pu->CTRLA=0;
}


void adc_init(void){

    ADCA.CALL = SP_ReadCalibrationByte( PROD_SIGNATURES_START +
ADCACAL0_offset );
    ADCA.CALH = SP_ReadCalibrationByte( PROD_SIGNATURES_START +
ADCACAL1_offset );


    ADCA.CTRLA = 0b00111101; // set up with a four channel sweep
    ADCA.CTRLB = 0b00010000; // signed
    ADCA.REFCTRL =0b00110000;     //portb reference, temp and
bandgap disabled
    ADCA.EVCTRL = 0b11000000;  // no events
    //ADCA.PRESCALER = 0b00000011; // div 32
    //ADCA.PRESCALER = 0b00000111; // div 512, 62.5kHz
    ADCA.PRESCALER = 0b00000101; // div 128, 250kHz
    //ADCA.PRESCALER = 0b00000110; // div 256, 125kHz

    ADCA.CH0.CTRL =0b10000011; // start channel 0, differential
w/gain 1
    ADCA.CH0.MUXCTRL = 0b00000011; // Differential POS=A0 and
NEG=A7, Battery Voltage
    ADCA.CH0.INTCTRL = 0b00000000; // no interrupt on channel 0;
```

```
    //ADCA.CH1.CTRL =0b10011011; // start channel 1, differential
w/gain 64
    //ADCA.CH1.CTRL =0b10010011; // start channel 1, differential
w/gain 16
    ADCA.CH1.CTRL =0b10001111; // start channel 1, differential
w/gain 8
    ADCA.CH1.MUXCTRL = 0b00010010;  // Differential POS=A2 and
NEG=A6, Battery Current
    //ADCA.CH1.MUXCTRL = 0b00110010;  // Differential POS=A6 and
NEG=A6, Battery Current
    ADCA.CH1.INTCTRL = 0b00000000; // no interrupt on channel 1;

    //ADCA.CH2.CTRL = 0b10011011; // start channel 2, differential
w/gain 64
    ADCA.CH2.CTRL = 0b10000011;   // start channel 2, differential
w/gain 1
    ADCA.CH2.MUXCTRL = 0b00100001; // Differential POS=A4 and
NEG=A5, MOSFET Current
    //ADCA.CH2.CTRL = 0b10000011;
    //ADCA.CH2.MUXCTRL = 0b00101001; // Differential POS=A4 and
NEG=A5, MOSFET Current
    ADCA.CH2.INTCTRL = 0b00000000; // no interrupt on channel 2;

    ADCA.CH3.CTRL = 0b10000001; // start channel 3, single-ended
    ADCA.CH3.MUXCTRL = 0b00011000; // Batt temp
    ADCA.CH3.INTCTRL = 0b00000001; // low interrupt on channel 3;
}


// ADC interrupt service routine
interrupt [ADCA_CH3_vect] void ADCA_CH3_isr(void){
    char i;
    //sleep_disable();

    // the ADC does not seem to bounds check properly, so I'll have
to do it
    ADCbuffer = ADCA.CH0.RES;
    ADCtemp[0] += bound(ADCbuffer, 0, 2047);

    ADCbuffer = ADCA.CH1.RES;
    ADCtemp[1] += bound(ADCbuffer, -2048, 2047);

    ADCbuffer = ADCA.CH2.RES;
    ADCtemp[2] += bound(ADCbuffer, -2048, 2047);

    ADCbuffer = ADCA.CH3.RES;
    ADCtemp[3] += bound(ADCbuffer, 0, 2047);

    ADCsamplecount++; //16 super sample
    if(ADCsamplecount > 63){


        //store the values
        ADCvalues[0] = ADCtemp[0]>>6;
```

```
        ADCvalues[1] = ADCtemp[1]>>6;
        ADCvalues[2] = ADCtemp[2]>>6;
        ADCvalues[3] = ADCtemp[3]>>6;

        for(i = 0; i<4; i++){
            ADCtemp[i] = 0;
        }

        ADCsamplecount = 0;
        //printf("%4d,
%4d\r\n",temp,battVolt,ADCvalues[1],ADCvalues[2]);
    }

    //Initiate next samples
    ADCA.CH0.CTRL |=0b10000000;
    ADCA.CH1.CTRL |=0b10000000;
    ADCA.CH2.CTRL |=0b10000000;
    ADCA.CH3.CTRL |=0b10000000;
    //sleep_enable();
    //idle();
}


interrupt [TWIC_TWIM_vect] void TWIC_TWIM_isr(void){
       //unsigned char x = 0;
       if ((TWIC.MASTER.STATUS &
TWI_MASTER_WIF_bm)>>TWI_MASTER_WIF_bp){

            if ((TWIC.MASTER.STATUS &
TWI_MASTER_ARBLOST_bm)>>TWI_MASTER_ARBLOST_bp){ //CASE M1
                //IF BUSY STATE -> MAKE IDLE
                TWIC.MASTER.STATUS = (TWIC.MASTER.STATUS &
~TWI_MASTER_BUSSTATE_gm) | TWI_MASTER_BUSSTATE_IDLE_gc;
            }else if((TWIC.MASTER.STATUS &
TWI_MASTER_RXACK_bm)>>TWI_MASTER_RXACK_bp){ //CASE M2
                #asm("wdr")

                // This code makes the master skip any cells that
are not responding... be sure to comment out "rewrite slave address
below" too
                 // if(/*slaveAddress==4 || slaveAddress==5 ||*/
slaveAddress==6){
                 //       battIdx++;
                 //       dataIdx=0;
                 //       slaveAddress=battIdx+1;
                 //       currAddress=slaveAddress<<1|READ;
                 //}else{
                    //PORTD.OUTCLR = 2;
                    //for (x=0; x<slaveAddress; x++){
                    //    delay_ms(300);
                    //    PORTD.OUTTGL = 4;
                    //    delay_ms(300);
                    //    PORTD.OUTTGL = 4;
                    //}
                    //delay_ms(1000);
```

```
                //}


                TWIC.MASTER.ADDR = currAddress;   //rewrite slave
address
                PORTD.OUTTGL = 4; //Toggle Red Led 1 if stuck.
            }else{  //MASTER WRITE SEQUENCE
               #asm("wdr")
               /*if(dataCnt==0){
                    dataCnt=1;
                    TWIC.MASTER.DATA = chargeStatus;
                    //TWIC.MASTER.DATA = dataOutTWIC;
               }else if(dataCnt==1){
                    dataCnt=2;
                    TWIC.MASTER.DATA = dataOutTWIC;
               }else{
                    TWIC.MASTER.CTRLC = (TWIC.MASTER.CTRLC &
~TWI_MASTER_CMD_gm) | TWI_MASTER_CMD_STOP_gc;
                    dataCnt=0;
               }*/
               if(dataCnt==0){
                    dataCnt=1;
                    TWIC.MASTER.DATA = dataOutTWIC;
               }else{
                    TWIC.MASTER.CTRLC = (TWIC.MASTER.CTRLC &
~TWI_MASTER_CMD_gm) | TWI_MASTER_CMD_STOP_gc;
                    dataCnt=0;
                    twiLock=0;
               }

            }

       }else if ((TWIC.MASTER.STATUS &
TWI_MASTER_RIF_bm)>>TWI_MASTER_RIF_bp){ //MASTER READ SEQUENCE
               battData[battIdx][dataIdx++] = TWIC.MASTER.DATA;
               if(dataIdx<10){
                    //Send Ack and Get Next Byte
                    TWIC.MASTER.CTRLC = (TWIC.MASTER.CTRLC &
~TWI_MASTER_CMD_gm & ~TWI_MASTER_ACKACT_bm) |
TWI_MASTER_CMD_RECVTRANS_gc;
               }else{
                    //Send Nack and Stop Condition
                    TWIC.MASTER.CTRLC = (TWIC.MASTER.CTRLC &
~TWI_MASTER_CMD_gm) | TWI_MASTER_CMD_STOP_gc |
TWI_MASTER_ACKACT_bm;
                    newData=1;
                    dataIdx=0;
                    PORTD.OUTCLR = 4; //Ensure RED LED 1 Off if
communicating
                    #asm("wdr")
               }
       }
}
```

```
void init_I2C_Mast()
{

    //000000,  BIT 7:2 RESERVED
    //0,  BIT 1 SDAHOLD, 1-ENABLE, 0-DISABLE
    //0, BIT 0 EXTERNAL DRIVER ENABLE, 1=EXTERNAL ENABLE, 0=NORMAL
TWI
    TWIC.CTRL = 0b00000000;

    //0000, BIT 7:4 RESERVED
    //00,   BIT 3:2 TIMEOUT, INACTIVE BUS TIMEOUT, 00=DISABLED
    //0,    BIT 1, QCEN, Quick Cmd enable
    //0,    BIT 0, SMEN, Smart mode enable, ACK sent after data
read
    TWIC.MASTER.CTRLB = 0b00000000;

    //00000, BIT 7:3 Reserved
    //1, BIT 2, ACKACT Acknowledge Action bit, 1=Send Ack, 0=send
Nack
    //0, BIT 1:0, CMD BITS, 00=RESERVED, 01=execute ack w/ repeated
start
    //                      10=execute ack w/ byte receive
    //                      11=execute ack w/ stop condition

    TWIC.MASTER.CTRLC = 0b00000100;


    //Force busstate to idle clear all other bits.
    TWIC.MASTER.STATUS = 0b11001101;

    TWIC.MASTER.ADDR=0;
    TWIC.MASTER.DATA=0;
    //TWMBR = fsys/(2*Ftwi)-5 = 32e6/(2*75e3)-5 = 208 == 75kHz baud
    //TWIC.MASTER.BAUD = 208;

    //TWMBR = fsys/(2*Ftwi)-5 = 32e6/(2*100e3)-5 = 155 == 100kHz
baud
    //TWIC.MASTER.BAUD = 155;

    //TWMBR = fsys/(2*Ftwi)-5 = 32e6/(2*25e3)-5 = 155 == 25kHz baud
    TWIC.MASTER.BAUD = 255;

    //TWMBR = fsys/(2*Ftwi)-5 = 32e6/(2*300e3)-5 = 48 == 300kHz
baud
    //TWIC.MASTER.BAUD = 48;

    //00, BIT 7:6, Interrupt Level, 11=HIGH, 10=MED, 01=LOW, 00=OFF
    //0, BIT 5, RIEN, Read Interrupt
    //0, BIT 4, WIEN, Write Interrupt Enable
    //0, BIT 3, ENABLE, TWI MASTER
    //0, BIT 2:0, RESERVED
    TWIC.MASTER.CTRLA = 0b01111000;

}
```

```
void main(void)
{
// Declare your local variables here

int movAvgValues[3][64];
long movAvgTotal[3]={0,0,0};
int newAdcVal[3];
unsigned int initSamples=0;
unsigned int sampleIdxOldest=0;

unsigned char n;
unsigned char ledVal;

int mosCurrRaw;
int battCurrRaw;
int battVoltRaw;
unsigned int socRaw=0x0700;
unsigned int tempRaw=0;

unsigned int i=0;

unsigned int duty=0;

float error = 0.0;
float refV = 0.0;
float KpV = 0;


unsigned char currBattIdx=0;
unsigned int chargerConnectedCnt=0;

KpV = -200;
refV = STOP_CHARGE_VOLT;

// Make sure the interrupts are disabled
#asm("cli")
// Low level interrupt: On
// Round-robin scheduling for low level interrupt: Off
// Medium level interrupt: On
// High level interrupt: On
// The interrupt vectors will be placed at the start of the
Application FLASH section
n=(PMIC.CTRL & (~(PMIC_RREN_bm | PMIC_IVSEL_bm | PMIC_HILVLEN_bm |
PMIC_MEDLVLEN_bm | PMIC_LOLVLEN_bm))) |
    PMIC_LOLVLEN_bm | PMIC_MEDLVLEN_bm | PMIC_HILVLEN_bm;
CCP=CCP_IOREG_gc;
PMIC.CTRL=n;
// Set the default priority for round-robin scheduling
PMIC.INTPRI=0x00;

// Watchdog timer initialization
watchdog_init();

// System clocks initialization
```

```
system_clocks_init();

// Event system initialization
event_system_init();

// Ports initialization
ports_init();

init_I2C_Mast();

// Virtual Ports initialization
//vports_init();

delay_ms(400);

// Timer/Counter TCC1 initialization
tcc1_init();

// Timer/Counter TCD0 is enabled, used to initiate talking to
boards.
tcd0_init();

// Timer/Counter TCD1 is enabled
tcd1_init(); //stepready

// Timer/Counter TCE0 initialization
//tce0_init();

// RTC initialization
rtcxm_init();

// USARTC0 is disabled
usart_disable(&USARTC0);

// USARTC1 is disabled
usart_disable(&USARTC1);

// USARTD0 is disabled
usart_disable(&USARTD0);

// USARTD1 is disabled
usart_disable(&USARTD1);

// USARTE0 is disabled
usart_disable(&USARTE0);

// SPIC initialization
//spic_init();

// ADC Initilization
adc_init();
usartC1_init();
usartE0_init();

ADCA.CH0.MUXCTRL = 0b00110010;  // Differential POS=A6 and NEG=A6
```

```
ADCA.CH1.MUXCTRL = 0b00110010;  // Differential POS=A6 and NEG=A6
ADCA.CH2.MUXCTRL = 0b00110010;  // Differential POS=A6 and NEG=A6

//Calibrate Differential Gain Offset
for (i=0; i<64; i++){
    ADCA.CH0.CTRL |=0b10000000;
    ADCA.CH1.CTRL |=0b10000000;
    ADCA.CH2.CTRL |=0b10000000;
    while((ADCA.CH2.INTFLAGS&0x01)!=1);
    ADCA.CH2.INTFLAGS |= 0x01;  //clear flag

    OFFSET_CH0_DIFF += ADCA.CH0.RES;
    OFFSET_CH1_DIFF += ADCA.CH1.RES;
    OFFSET_CH2_DIFF += ADCA.CH2.RES;
}

OFFSET_CH0_DIFF = OFFSET_CH0_TOTAL>>6;
OFFSET_CH1_DIFF = OFFSET_CH1_TOTAL>>6;
OFFSET_CH2_DIFF = OFFSET_CH2_TOTAL>>6;

ADCA.CH0.MUXCTRL = 0b00000011; // Differential POS=A0 and NEG=A7,
Battery Voltage
ADCA.CH1.MUXCTRL = 0b00010010; // Differential POS=A2 and NEG=A6,
Battery Current
ADCA.CH2.MUXCTRL = 0b00100001; // Differential POS=A4 and NEG=A5,
MOSFET Current


// Globaly enable interrupts
#asm("sei")


//Battery Voltage Control
KpV = -100;
refV = STOP_CHARGE_VOLT;

watchdog_init();
desiredCurrentInt=desiredCurrentIntEE;
desiredCurrentDec=desiredCurrentDecEE;

    while (1)
    {
          if(stepready)
          {

                stepready=0;
                newAdcVal[0] = ADCvalues[0]; //Raw Batt Voltage
                newAdcVal[1] = ADCvalues[1]; //Raw Batt Current
                newAdcVal[2] = ADCvalues[2]; //Raw MOSFET Current

                if(initSamples<64) //Grabs first 128 samples to
start off average filter, this all occurs during INIT MODE so
battVoltageFilt can be wrong during this time
                {
                   for (i=0; i<3; i++)
```

```
                    {
                         movAvgValues[i][initSamples] =
newAdcVal[i];
                         movAvgTotal[i] = movAvgTotal[i] +
movAvgValues[i][initSamples]; //Eventually will hold total of first
128 samples
                    }
                    initSamples++;
                }
                else
                {
                    for (i=0; i<3; i++)
                    {
                         movAvgTotal[i] = movAvgTotal[i] -
movAvgValues[i][sampleIdxOldest]; //Subtracts out oldest sample
                         movAvgValues[i][sampleIdxOldest] =
newAdcVal[i]; //Replace oldest sample value with a new value
                         movAvgTotal[i] =
movAvgTotal[i]+movAvgValues[i][sampleIdxOldest]; //Add in newest
value, movAvgTotal is now last 128 values, increment oldest index
                    }
                    sampleIdxOldest++;
                    sampleIdxOldest&=0x3F; //Wraps around Oldest
Index value in case it "overflows" out of range of movAvgValues[]
array
                }

               battVoltRaw = movAvgTotal[0]>>6;
               battCurrRaw = movAvgTotal[1]>>6;
               mosCurrRaw = movAvgTotal[2]>>6;
               tempRaw = ADCvalues[3];

               offset = (signed int)(ADC_OFFSET_M[0]*mosCurrRaw +
ADC_OFFSET_B[0]);
               mosCurrRaw = mosCurrRaw+offset;

               offset = (signed int)(ADC_OFFSET_M[0]*battVoltRaw +
ADC_OFFSET_B[0]);
               battVoltRaw = battVoltRaw+offset;

               battVolt[0] =  battVoltRaw*0.0025;
               mosCurr[0] = mosCurrRaw*.001; //1 ohm, gain 1
               battCurr[0] = battCurrRaw*0.020833333; //6 mOhm,
gain 8


               if (mosCurr[0]<0) mosCurr[0]=0;
               temp[0] = -1481.96 + sqrt(2196200 + (1863.9-
tempRaw)*257.732);
               socRaw = (unsigned
int)((float)4095.0*(float)((float)AmpSec/(float)MAX_AMP_SEC));


               soc[0] = socRaw*0.0244140625;
```

```
                if(AmpSec<=0) AmpSec=0;

                 //Loop occurs at 4.32ms with printf statement

                 if(mosCurr[0]>MAX_BYPASS_CURRENT)
                 {
                     ledColor=RED;
                     byPassMax[0]=1;
                 }
                 else if(mosCurr[0]>0.05)
                 {
                     ledColor=AMBER;
                     byPassOn[0]=1;
                 }
                 else
                 {
                     ledColor=GREEN;
                     byPassOn[0]=0;
                     byPassMax[0]=0;
                 }

                 if(battVolt[0]<OVERDISCHARGE_VOLT)
overDischarge[0]=1;
                 else overDischarge[0]=0;

                 voltCntrlCnt++;
                 if(voltCntrlCnt>75)
                 {
                     voltCntrlCnt=0;
                     error = (refV-battVolt[0]);
                     if (error>=0)   //if batteryVoltage <3.8 means
error is positive
                     {
                         if((duty+error*KpV)<=0) //prevent
controller from causing duty from rolling over
                         {
                             duty=0;
                         }
                         else
                         {
                             duty=duty+KpV*error;
                         }
                     }
                     else
                     {
                         duty=duty+KpV*error; // +
KpV*250*integral_V;
                     }
                 }

                //Cap duty cycle to a maximum value.
                if(duty>DUTY_CYC_MAX) duty=DUTY_CYC_MAX;

                    //Ensure mosfet stays off if battery is not
charging and battery voltage is less than max
```

```
                if(battCurr[0]<=0.05 &&
battVolt[0]<(STOP_CHARGE_VOLT))
                {
                     if(dutyOffCnt<=MAX_DUTY_OFF_CNT) dutyOffCnt++;

                     if(dutyOffCnt>=MAX_DUTY_OFF_CNT)
                     {
                         TCC1.CCA=0;
                     }
                }
                else
                {
                     TCC1.CCA=duty;
                }

           }

            if(dataOutTWIC == LOW_SOC)
            {
                 ledVal=LOW_SOC;
            }
            else
            {
                 if(dataOutTWIC>6)
                 {
                     ledVal=7;
                 }
                 else
                 {
                    ledVal=dataOutTWIC;
                    if(ledVal>6)ledVal=7;
                 }
            }

            switch(ledVal)
            {
                 case 1:
                     PORTD.OUT = 0;
                     PORTE.OUT = 0;
                     PORTD.OUTSET = ledColor; //0b00000010==2
                     break;
                 case 2:
                     PORTD.OUT = 0;
                     PORTE.OUT = 0;
                     PORTD.OUTSET = ledColor<<2; //0b00001000==8
                     break;
                 case 3:
                     PORTD.OUT = 0;
                     PORTE.OUT = 0;
                     PORTD.OUTSET = ledColor<<4; //0b00100000==32
                     break;
                 case 4:
                     PORTD.OUT = 0;
                     PORTE.OUT = 0;
                     PORTD.OUTSET = ledColor<<6; //0b10000000==128
```

```
                    break;
                case 5:
                    PORTD.OUT = 0;
                    PORTE.OUT = 0;
                    PORTE.OUTSET = 0b0011 & ledColor;
//0b00000010==2
                    break;
                case 6:
                    PORTD.OUT = 0;
                    PORTE.OUT = 0;
                    PORTE.OUTSET = 0b0011 & (ledColor<<2);
//0b00001000==8
                    break;
                case LOW_SOC:
                    PORTD.OUTTGL = 0b01010101;
                    PORTE.OUTTGL = 0b00000001;
                    break;
                default:
                    PORTD.OUT = ledColor;
                    PORTE.OUT = 0;
                    break;
           }

          //GRAB DATA FROM OTHER ICMU SLAVES
          currBattIdx=battIdx; //HOLD BATTERY CELL INDEX VALUE
WHILE RUNNING THE FOLLOWING LOOP IN CASE battIdx IS CHANGED IN
INTERRUPT
          if(newData)
          {
                if(currBattIdx!=0) //IF NOT THE MASTER, WE NEED TO
CONVERT MEASURMENT DATA FROM PACKET BACK TO ACTUAL VALUE
                {
                    battCurrRaw = (signed int)((((unsigned
int)battData[currBattIdx][1])<<8) | battData[currBattIdx][0]);
                    battVoltRaw = (signed int)((((unsigned
int)battData[currBattIdx][3])<<8) | battData[currBattIdx][2]);
                    mosCurrRaw = (signed int)((((unsigned
int)battData[currBattIdx][5])<<8) | battData[currBattIdx][4]);
                    tempRaw = (unsigned int)((((unsigned
int)battData[currBattIdx][7])<<8) | battData[currBattIdx][6]);

                    //NEED TO GRAB STATUS BITS FROM PACKET
                    byPassOn[currBattIdx]=(battData[currBattIdx][9]
& 0x80)>>7;

byPassMax[currBattIdx]=(battData[currBattIdx][9] & 0x40)>>6;
                    overDischarge[currBattIdx] =
(battData[currBattIdx][9] & 0x20)>>5;
                    overTemp[currBattIdx] =
(battData[currBattIdx][9] & 0x10)>>4;

                    //GRAB SOC DATA FROM CURRENT CELL INDEX SOC
                    battData[currBattIdx][9] =
battData[currBattIdx][9]&0x0F;
```

```
                    socRaw = (unsigned int)((((unsigned
int)battData[currBattIdx][9])<<8) | battData[currBattIdx][8]);

                    //CONVERT RAW DATA TO SCIENTIFIC NOTATION
                    battVolt[currBattIdx] =  battVoltRaw*0.0025;
                    mosCurr[currBattIdx] = mosCurrRaw*.001; //1
ohm, gain 1
                    battCurr[currBattIdx] =
battCurrRaw*0.020833333; //6 mOhm, gain 8
                    temp[currBattIdx] = tempRaw*0.01;
                    soc[currBattIdx] = socRaw*0.02442;
                }

                //PRINT OUT CURRENT CELL'S DATA ONLY IF THE BPMU'S
MEASURED CURRENT IS ABOVE A CERTAIN THRESHOLD
                if(battCurr[0]>0.08 || battCurr[0]<-0.08)
                {

printf("%05u,%02d,%03.2f,%03.3f,%+03.3f,%+03.2f,%03.2f,%1d,%1d,%1d,
%1d\r\n",RTC.CNT,currBattIdx+1,temp[currBattIdx],battVolt[currBatt
Idx],battCurr[currBattIdx],mosCurr[currBattIdx],soc[currBattIdx],byP
assOn[currBattIdx],byPassMax[currBattIdx],overDischarge[currBattIdx
],overTemp[currBattIdx]);
                }

                //CLEAR FLAG FOR newData AND WAIT FOR NEXT I2C
MESSAGE, INCREMENT INDEX TO THE NEXT ICMU
                newData=0;
                battIdx++;

                //CHECK TO SEE IF LAST ICMU'S INFORMATION HAS BEEN
RECEIVED.
                if(battIdx>=MAX_BATT_CNT)
                {
                        battIdx=0;
                        dataRdy=1;

                        //CHECK IF ANY CELL IS IN AND UNDERVOLT
CONDITION
                        underVoltCond = 0;
                        for(i=0;i<MAX_BATT_CNT;i++)
                        {
                           if(battVolt[i]<2.5 && soc[i]<0.1)
                           {
                                underVoltCond = 1;
                                break;
                           }
                           underVoltCond = 0;
                        }

                        //COUNTER FILTER FOR DETECTING UNDERVOLTAGE
CONDITION
                        if(underVoltCond)
                        {
```

```
                            if(underVoltCondCnt<65000)
underVoltCondCnt++;
                        }else
                        {
                            underVoltCondCnt=0;
                        }

                        //ENSURE THAT CONDITION EXISTS FOR A
CERTAIN COUNT BEFORE SETTING CONDITION
                        if(underVoltCondCnt>1000)
                        {
                            //SEND OUT LOW SOC MESSAGE TO ALL ICMUs
                            //ALL ICMUs WILL FLASH RED THEN
                            dataOutTWIC=LOW_SOC;
                        }
                        else //NOT LOW SOC DETECTED,
INCREMENT/DECREMENT NEXT NUMBER FOR "KNIGHT RIDER EFFECT"
                        {
                            if(increment)
                            {
                                dataOutTWIC++;
                            }
                            else
                            {
                                dataOutTWIC--;
                            }
                            if(dataOutTWIC>=MAX_BATT_CNT*6)
                            {
                                increment=0;
                            }else if (dataOutTWIC==0){
                                increment=1;
                            }
                        }

                        //Writing current count to all slaves for
cycling LED display pattern, data being written out is in
dataOutTWIC
                        currAddress=allCallAddress<<1|WRITE;
                        TWIC.MASTER.ADDR = currAddress;
                }
          }

          //ONCE ALL ICMU DATA HAS BEEN RECEIVED PRINT OUT
INFORMATION ON BATTERY PACK AS WHOLE.
          //SPECIFICALLY, PACK VOLTAGE, CURRENT, AND SOC (ZEROS ARE
JUST PLACEHOLDERS).
          if(dataRdy)
          {
                dataRdy=0;
                packCurr=0;
                packSoc=100.0;
                packVolt=0;
                //CALCULATE PACK VOLTAGE FROM SUM OF ALL ICMU's
CELL VOLTAGE
                for(i=0; i<MAX_BATT_CNT; i++)
```

```
                {
                    packCurr=battCurr[i]+packCurr; //SUM CURRENT
FOR CURRENT AVERAGE
                    packVolt=battVolt[i]+packVolt;
                    if(soc[i]<packSoc) packSoc=soc[i];
                }
                packCurr=packCurr/MAX_BATT_CNT;  //AVERAGE CURRENT
FOR ALL CELLS (SHOULD BE SAME)

printf("0,0,0,0,0,0,0,0,0,0,0,%03.3f,%+03.3f,%03.2f\r\n",packVolt,p
ackCurr,packSoc);
          }

          if(waitToChangeCnt<65530)
          {
            waitToChangeCnt++; //Prevent Rollover
          }



          //************** BEGIN LOGIC FOR CHARGER
CONTROL******************/

          //DETECT THAT CHARGER IS CONNECTED
          if(((PORTE.IN&PIN2_bm)>>PIN2_bp)==0) //Charger present if
pin is low
          {
                //FLASH LED 2 TIMES AMBER IF CHARGER IS DETECTED
                if(chargerConnectedCnt>2000)
                {
                    chargerConnectedCnt = 0;
                    PORTE.OUTTGL = 0b0011 & AMBER; //0b00000010==2
                    delay_ms(500);
                    PORTE.OUTTGL = 0b0011 & AMBER; //0b00000010==2
                    delay_ms(500);
                    PORTE.OUTTGL = 0b0011 & AMBER; //0b00000010==2
                    delay_ms(500);
                    PORTE.OUTTGL = 0b0011 & AMBER; //0b00000010==2
                    delay_ms(500);
                }


                //****CHECK: IS AN ICMU AT ITS MAXIMUM BYPASS
CURRENT?
                //LOGIC TO DETERMINE IF ANY ICMU IS AT MAX BYPASS
                battBypassId=100;
                for(i=0;i<MAX_BATT_CNT;i++)
                {
                    //BREAK OUT OF LOOP WHEN MAX IS DETECTED
                    //KEEPING THE ICMU'S INDEX THAT IS AT MAX
                    //IF NO CELL IS OVER BYPASS MAX CURRENT
                    //THEN battBypassId REMAINS 100 WHICH NO
                    //VALID ICMU WILL HAVE THAT VALUE IN THIS
SYSTEM
                    if(mosCurr[i]> MAX_BYPASS_CURRENT )
```

```
                    {
                        battBypassId = i;
                        break;
                    }
                }


                //IF battBypassId IS NOT 100, AND THE SAME VALUE
TWICE IN A ROW
                //START COUNT FOR DECREASING CHARGER CURRENT
                if ((battBypassId == prevBattBypassId) &&
(battBypassId != 100))
                {
                    if(decreaseCurrentCnt<8000)
                    {
                        decreaseCurrentCnt++;
                    }
                }
                else
                {
                    decreaseCurrentCnt=0;
                }


                //STORE PREVIOUS BYPASS ID FOR USE NEXT TIME AROUND
                prevBattBypassId=battBypassId;


                //****CHECK: ARE ALL BYPASSES ON?
                //LOGIC TO DETECT THAT ALL ICMUs ARE BYPASSING SOME
CURRENT
                allBypassOn=1;
                for(i=0;i<MAX_BATT_CNT;i++)
                {
                    allBypassOn &= (mosCurr[i]>0.05); //If
bypassing some current, means it's at 3.8
                }

                //IF ALL BYPASSES ARE ON, SETUP BMS MASTER MESSAGE
                //TO CHARGER TO SHUT-OFF
                if(allBypassOn){
                    if(turnOffChargerCnt < MAX_TURNOFF_CNT){
                        turnOffChargerCnt++;
                    }
                    else
                    {
                        turnOffChargerCnt=0;
                        desiredCurrentInt=0;
                        desiredCurrentDec=0;
                        sendDesiredCurrent=1;
                        waitToChangeCnt=0;
                    }
                }

                //****CHECK: ANY CELL OVERVOLTAGE (>4.2V)
```

```
                //LOWER CHARGING CURRENT TO 1.8 AMPS IF ANY CELL IS
DETECTED
                //AS BEING HIGHER THAN 4.2V FOR A DETERMINED PERIOD
OF TIME.
                //NOTE: ICMUs CAN BYPASS 1.9 AMPS
                overVoltageShutoff=0;
                for(i=0;i<10;i++)
                {
                    overVoltageShutoff |= (battVolt[i]>4.2);
                }

                //COUNTER FOR FILTERING SHUT-OFF LOGIC FROM NOISE
IN MEASUREMENTS
                if(overVoltageShutoff)
                {
                    if(turnOffChargerCnt2 < MAX_TURNOFF_CNT)
                    {
                        turnOffChargerCnt2++;
                    }
                    else
                    {
                        turnOffChargerCnt2=0;
                        desiredCurrentInt=2;
                        desiredCurrentDec=2;
                        sendDesiredCurrent=1;
                        waitToChangeCnt=0;
                    }
                }

                //****CHECK: ANY CELL OVERTEMPERATURE
                overTemperatureShutoff=0;
                for(i=0;i<10;i++)
                {
                    overTemperatureShutoff |= overTemp[i];
                }

                //COUNTER FOR FILTERING SHUT-OFF LOGIC FROM NOISE
IN MEASUREMENTS
                if(overTemperatureShutoff)
                {
                    if(turnOffChargerCnt3 < MAX_TURNOFF_CNT)
                    {
                        turnOffChargerCnt3++;
                    }
                    else
                    {
                        turnOffChargerCnt3=0;
                        desiredCurrentInt=0;
                        desiredCurrentDec=0;
                        sendDesiredCurrent=1;
                        waitToChangeCnt=0;
                    }
                }
```

```
                //IF COUNTERS EXCEED AND ARE NOT RESET BY NORMAL
OPERATION FROM CHECKS ABOVE
                //CREATE MESSAGE TO CHARGER TO REDUCE CHARGING
CURRENT

                if(decreaseCurrentCnt>5000 &&
waitToChangeCnt>WAIT_BEFORE_CHANGE_CNT)
                {
                    //IF CHARGING CURRENT IS 6 OR GREATER DECREASE
CURRENT BY 2
                    if(desiredCurrentInt>=6)
                    {
                        desiredCurrentInt=desiredCurrentInt-2;
                        desiredCurrentDec=0;
                    }
                    else //OTHERWISE, SET CURRENT TO 1.8 A
                    {
                        desiredCurrentInt=1;
                        desiredCurrentDec=8;
                    }
                    sendDesiredCurrent=1;
                    waitToChangeCnt=0;
                    decreaseCurrentCnt=0;
                }

                //SEND MESSAGE TO CHARGER
                if(sendDesiredCurrent==1)
                {
                    sendDesiredCurrent=0;

                    //CHECK THAT THE NEW VALUE IS NOT THE SAME AS
WHAT IS CURRENTLY
                    //IN THE BPMU'S EEPROM IF IT IS NEW, STORE IT
IN BPMU's MEMORY
                    if(desiredCurrentIntEE != desiredCurrentInt)
                    {
                        desiredCurrentIntEE = desiredCurrentInt;
                        desiredCurrentDecEE = desiredCurrentDec;
                    }

                    //CREATE PACKET
                    //BYTE0: 0xAA IS START OF PACKET INDICATOR
                    //BYTE1: DESIRED CURRENT INTEGER PORTION
                    //BYTE2: DESIRED CURRENT DECIMAL PORTION
                    //BYTE3: DESIRED CURRENT INTEGER PORTION
                    //BYTE4: DESIRED CURRENT DECIMAL PORTION
                    //0x55 IS END OF PACKET INDICATOR
                    //BYTES 1&2 ARE REDUNDANT WITH BYTE 3&4 FOR
TRANSMISSION ERROR CHECKING
                    //ON CHARGER SIDE OF TRANSMISSION
                    currentPkt[0]=0xAA;
                    currentPkt[1]=desiredCurrentInt;
                    currentPkt[2]=desiredCurrentDec;
                    currentPkt[3]=desiredCurrentInt;
                    currentPkt[4]=desiredCurrentDec;
```

```
                    currentPkt[5]=0x55;

                    //SEND SAME PACKET 5 TIMES IN A ROW FOR CHARGER
                    //IF CHARGER DOES NOT SEE THE SAME MESSAGE 5
                    //IT WILL NOT CHANGE VALUE
                    for(i=0;i<6;i++) uartE0_putchar(currentPkt[i]);
                    for(i=0;i<6;i++) uartE0_putchar(currentPkt[i]);
                    for(i=0;i<6;i++) uartE0_putchar(currentPkt[i]);
                    for(i=0;i<6;i++) uartE0_putchar(currentPkt[i]);
                    for(i=0;i<6;i++) uartE0_putchar(currentPkt[i]);
                }

          }
          else //SYSTEM IS IN "CHARGER IS NOT CONNECTED" STATE
          {
                //NOISE FILTER: chargerConnected STARTS TO COUNT UP
WHEN CHARGER IS DETECTED
                //THIS VALUE IS RESET ABOVE WHEN THE CHARGER
DETECTION PIN IS LOW
                if(chargerConnectedCnt<3000)
                {
                    chargerConnectedCnt++;
                }
                //IF CHARGER IS DETECTED, SEND A MESSAGE TO CHARGER
                //TO TELL IT TO START CHARGING AT 10 AMPS
                if(chargerConnectedCnt>2000)
                {
                    desiredCurrentInt=10; //Integer
                    desiredCurrentDec=0;  //Decimal Value

                    //CHECK IF NEW CHARGING CURRENT IS THE SAME AS
WHAT
                    //IS IN BPMU'S EEPROM MEMORY
                    if(desiredCurrentIntEE != desiredCurrentInt)
                    {
                        //STORES 10A CHARGING CURRENT IN BPMU'S
MEMORY
                        //IN THE EVENT OF A RESET OCCURING
                        desiredCurrentIntEE = desiredCurrentInt;
                        desiredCurrentDecEE = desiredCurrentDec;
                    }
                    sendDesiredCurrent=1; //TELL CHARGER
COMMUNICATION LOGIC TO SEND A MESSAGE
                }
          }
     }

}
```