Faculty of Engineering of University of Porto

Development of a battery management system interface for adapting second life batteries to energy storage

Jorge Lopes dos Santos Teixeira Carrondo

Thesis carried out in the
Integrated Master in Electrical and Computers Engineering
Energy Major

Supervisor(FEUP): Rui Esteves Araújo
Supervisor(New Electric): Celso Menaia

February 2019

Abstract

With the advent of electric vehicles, surges the issue of the used batteries for vehicles, namely cars, such as Tesla (Models X and S), VW (GTE, E-Golf and E-Up), Nissan (Leaf and NV200) and BMW i3. After its normal daily use, the batteries might not be suited anymore to be utilized in a different vehicle since it requires power peaks. However, there is still an important and ever-growing use case for these batteries as an energy storage unit that can be used for domestic and industrial installations paired up with renewable energy injection.

In order to safely and accurately provide energy there is a need of technical and theoretical expertise not only for installing the system but to regulate it too. For that, an interface that can communicate directly with batteries and respective BMS is a breakthrough.

This project aims to study the battery management system used in electric vehicles and obtain as much information as possible regarding its communication protocol and control features. Based on the information obtained by using SavvyCAN and reverse engineering methods to infer how the batteries communicate, it will be developed an electronic interface with a Teensy Arduino board tailored for this specific usage, in order to adapt the second life batteries from electric vehicles to a possible storage solution, namely, an interface, using OPUS Projektor, to visualise and control each and every paramenter necessary, in conjunction with renewable energy injection.

Contents

List of figures

List of tables

List of equations

Abbreviations and symbols

List of abbreviations

AC	Alternated Current
BMCU	Battery Management Control Unit
BMS	Battery Management System
BPU	Battery Protection Unit
CAN	Controller Area Network
DC	Direct Current
DER	Distributed Energy Resources
DOD	Depth of Discharge
DSO	Distribution System Operator
ES	Energy Storage
EV	Electric Vehicle
FET	Field Effect Transistor
IEA	International Energy Agency
LC	Load Controller
LMU	Local Management Unit

MC	Microsource Controller
MGCC	Micro Grid Central Controller
OEM	Original Equipment Manufacturer
PV	Photovoltaic
RES	Renewable Energy System
s-BMS	Scalable BMS
SAO	Safe Area of Operation
SOC	State of Charge
SOH	State of Health
TSO	Transmission System Operator

List of units

Ah	Ampere Hour (Unit of charge)
Bd	Baud (Unit of speed in telecommunications)
C	Farad (F)
Hz	Hertz (Frequency)
kWh	KiloWatt Hour
Ω	Ohm (Resistance)
Φ	Phi (Phase angle)

Chapter 1

Introduction

The following chapter will give a simple and concise explanation on the basic topics to be discussed throughout the document as well as the motivation and context of the project in question. Thus, this is the first approach to the subject at hand in a scientific and technological perspective.

1.1 Motivation

As technology progresses so does the need to adapt and repurpose components and practices that need to be improved for the sake of efficiency, environmental protection, financial sustainability and to attain the best final product. With the advent of electric vehicles surges the issue of the used batteries for vehicles, namely cars such as Tesla (Models X and S), VW (GTE, E-Golf and E-Up), Nissan (Leaf and NV200) and BMW i3. After its normal daily use, the battery might not be suited anymore to be used again in a different vehicle since it requires power, however, there is still an important and ever-growing use case for said battery as an energy storage unit for domestic and industrial installations paired up with renewable energy injection.

This means that, in conjunction with a renewable energy unit, such as a solar panel, a wind turbine or a mini-hydric that may produce energy while it is not being used, a storage unit is able to mitigate that issue by saving unused energy for later use. These storage units can be bulky and expensive at times however, with the repurpose of electric vehicles' batteries; a solution can be obtained that is just as efficient with less space necessary and reduced costs. This will drive up the demand of storage units in the near future.

Finally, there are also waste and pollution concerns that come at play with recycling batteries. By extending their usage, the strain caused on the environment will be reduced.

The following thesis will focus on the problems and methods on how to convert as well as improve the efficiency of second life batteries, more specifically, better and more accurate readings (temperature, voltage, current…) and individualised control of the modules that contain them by assessing their characteristics and streamlining their functioning with a microcontroller (Arduino Due). This project occurred at New Electric in Amsterdam, a company which specialises in converting vehicles to electric and currently has a partnership to elaborate a storage unit prototype using batteries, either from recalls or electric vehicles that are already used. This means it is a project with industry and market capability and potential.

There is not only an enormous potential for storage facilities and equipment but also a deeply rooted need for it in order to make renewable energy sources more endemic and ubiquitous in the near future since, as previously stated, they cannot be easily controlled due to being subject to weather/climate conditions. Energy storage is then the missing link in production by means of renewable sources and daily usage since it allows for continuous use on a large scale independent of the time of day (day or night), weather conditions (wind, rain, sun) and possible system failures altogether. Electric vehicles are also a growing trend and mixing storage and EV (Electrical Vehicle) charging/discharging process in harmony will inevitably lead to a more feasible penetration of RES (Renewable Energy Sources) in the day to day life of the average consumer and industries, either big or small, alike.

To sum up, both EVs and energy storage can be used in sync as to compliment and round up the gaps in energy production and irregular consumption with the final objective of detachment from the electrical grid and independence from it without compromising efficiency and reliability.

1.2 Objectives

This project aims to study the batteries of EVs, in this particular case a VW and Tesla, and provide as much information regarding its communication protocol by means of reverse engineering and trial and error methodology in terms of connections and properties. The following step is to improve said communications and funnel them in order to both manage individual control and provide a tool to yield usability for other purposes. This can be achieved by microcontroller specifically built for this purpose, a Teensy Arduino board with open source coding on Github. Finally, I will build on the schematics and electronics as well as program in order to adapt the modules for large scale energy storage. It was heavily focused on data protocols and communication between batteries, a topic I learned throughout the project.

Throughout the document a market analysis will be present in order to better understand the necessity and demand of energy and the use case for storage.

The main topics this document will be focused on are:

- Analyse and improve communication protocols between different modules of the units with an Arduino microcontroller;
- Adapt electric vehicles' batteries after their normal functioning period for energy storage in both domestic and industrial installations;
- Increase usability and ease of access of said units by creating an interface for it;
- Constant improvement of said interface as an efficient and reliable product;
- Analysis and validation of test results on the final product.

1.3 Structure

The following document will be divided into 5 distinct sections, explained below.

In Chapter 1, Introduction, a simple description of the context and current environment of the technology and science being worked upon is presented as well as objectives of the project.

In Chapter 2, State of the Art, it will be given a presentation of current models, technologies and forecast of market sales, consumption needs and production goals.

In Chapter 3, Interface development, it will be presented the main specifications for the product, an overview of the connection that was established as well as the devices necessary for the interface. Then, the programming section of the microcontroller will be explained and, finally, how a possible integration with the grid can be done (brief explanation of the process).

Chapter 4, Practical cases and testing, will go through the setup of a Tesla module and VW set of modules and how the electronics will be changed to account for a different type of usage. Testing will be presented and explained.

Chapter 5, Conclusions, presents the final take and elaborates on the implementation feature and plan for real life use case and work which can be performed later on.

Chapter 2

State of the Art

Throughout this section, the present state of the technology and everything involved with its production and propagation will be analysed. This means the technology, markets and R&D cases will be targeted. A special focus will be given to BMS (Battery Management System) structure and how it operates as well as concrete issues it solves.

2.1 Current use case

OEM batteries are an essential part of the ever-growing market of the electric and sustainable new paradigm of transportation either by land, in the form of cars and other vehicles such as efficient trains, trucks and motorcycles but also maritime transport of people (ferryboats and passenger boats) but also big cargo ships. In the near future, there is also possibility of electric or hybrid aeroplanes, being researched by companies like Rolls-Royce [1].

The sale of electric cars has been steadily increasing with more than 1 million being sold in 2017 [2] with projection of 125 million units by the year of 2030, according to the IEA (International Energy Agency). It is safe to say, then, that there is a considerable demand for and development of electric vehicles. Companies such as Tesla are at the forefront of development in batteries and charging solutions for both vehicles and also home facilities with their Powerwall model, Figure 1, which can supply a home for over a week (depending on size and energetic needs) even without any connection to the electrical grid [3], which means more independence and control to the end-user.

Figure 1 - Inside of a Tesla Powerwall unit [3]

This unit is scalable up to 10 units and has internal converter (bidirectional DC/DC) and connection to the grid in the form of DC/AC, as well as embedded cooling unit and connection point. An energy storage unit usually gets its input from a RES such as photovoltaic panels or wind turbines which, by default, are not able to be controlled by human necessity due to being subject to weather conditions. That is one of the main reasons why storing energy while it is not being used is important in this new paradigm of clean and renewable sources of energy.

According to the IEA, there are still long strides to be made in the energy storage field. The value of storage in 2017 was 15300 MWh [4]. This was divided into 39% of compressed air storage, 28% related to lithium-ion batteries, 12% to flow batteries and 6% to sodium-based ones with other technologies covering the remainder.

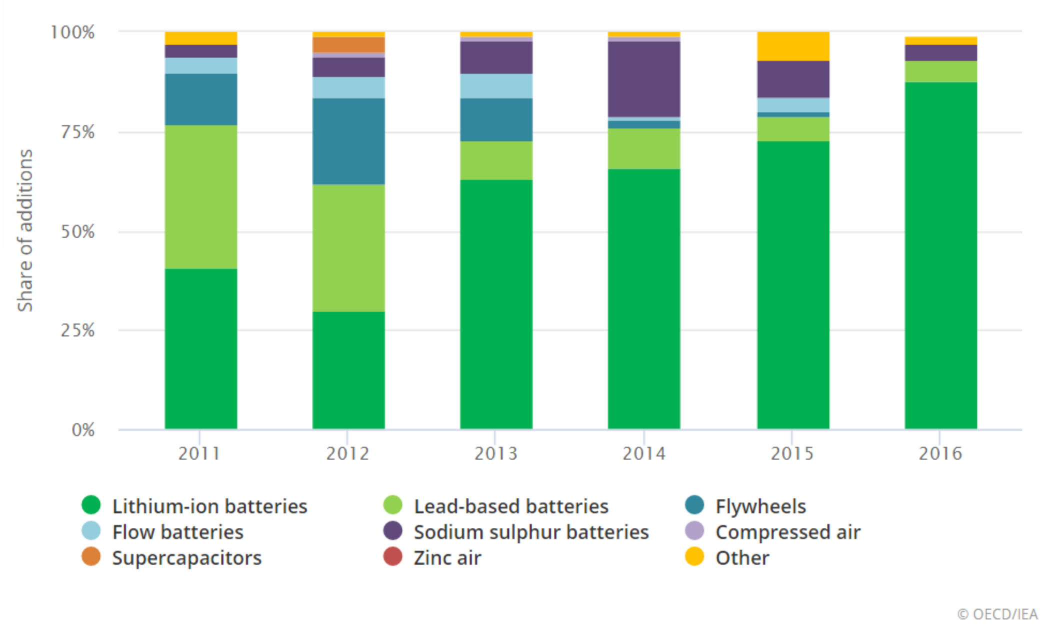


Figure 2 - Share of annual battery storage additions, by technology [4]

By analysing the histogram above from Figure 2, it is safe to say that lithium-ion batteries are the dominant and ever-growing part of energy storage, creating a monopoly when it comes to that technology in the upcoming years. Battery costs continue to drop, as unit costs for lithium-ion decreased 22% between 2016 and 2017. While prices of active materials increased substantially over 2017, particularly those of cobalt (which more than doubled) and lithium (which grew by half), they have not had a substantial effect on battery prices [4].

Storage facilities provide a reliable and secure alternative to the mainstream electrical grid as a new paradigm of self-sustainability and independence. This change of structuring in the electrical system has advantages in multiple fronts:

- Technologically it provides more reliability by decentralizing the production of energy meaning it's less fallible to system failures and weather unpredictability (clouds in a centralised solar power plant can cause lengthy loss of energy production);
- On an economical perspective it diversifies the energy market making it healthier;
- Independent fiscal benefits either by grants and subsidies for renewable energy but mostly due to becoming a “prosumer” (consumer who produces own energy);
- Environmental reasons due to reducing fossil fuel usage and energy waste as well as reusing pollutant lithium batteries.

2.2 BMS review

The BMS of a module is an integral part of its control capabilities in terms of regulating voltage, temperature, current and other factors. It is essential for its safe usage by guaranteeing its constant functioning in the SAO (Safe Area of Operation), gathering and providing data and, eventually, improving and optimising the battery.

The architecture of a BMS is quite complex, as can be visualised in Figure 3, and it involves logic programmable units with blocks that regulate the current it passes through it by means of cutoff with electronic systems. A FET (Field Effect Transistor), linked to the charger, is able to cut the transit of current between the charger and battery in case of over temperature or a short-circuit situation in order not to damage the battery or the load.

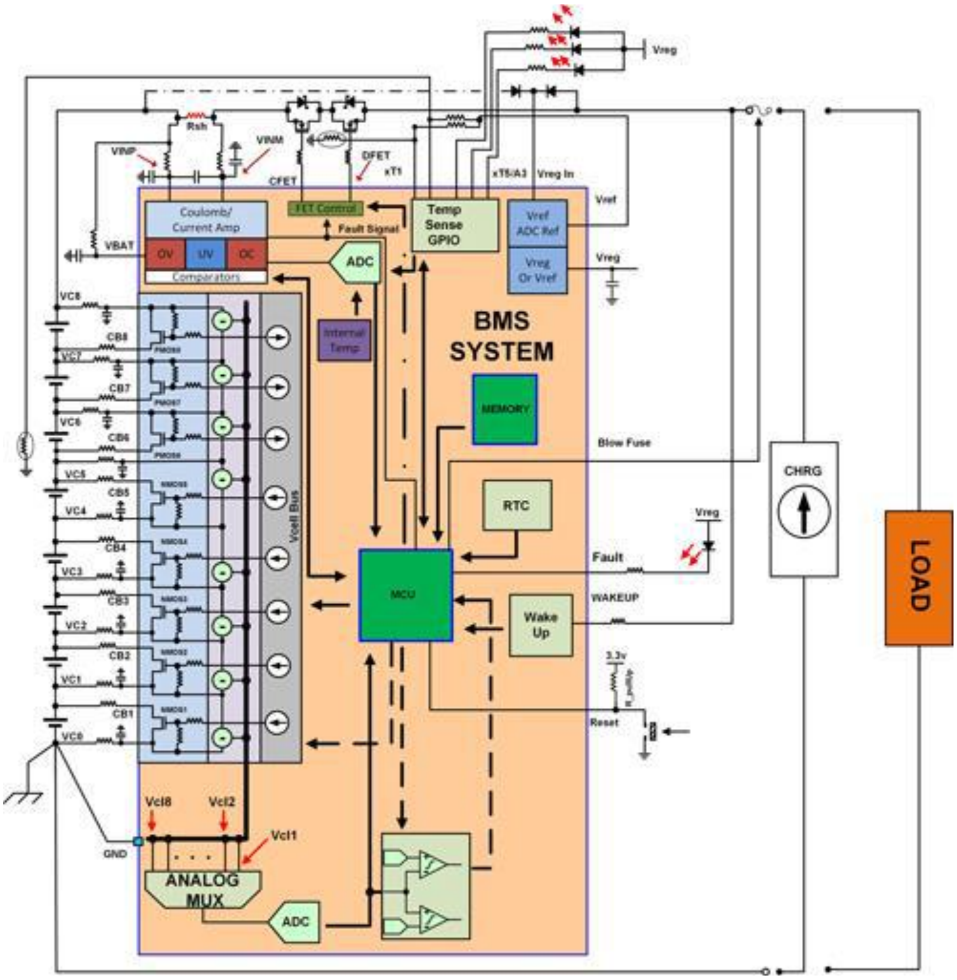


Figure 3 - Schematic of a BMS [5]

Several programmable units are embedded in a BMS unit as well as multiplexers. Temperature sensors are also essential in conjunction with circuit breakers and converters. The BMS is the core of the battery control as it will be detailed further along this document.

SOC (State of Charge), SOH (State of Health) and other equations are essential to understanding how a battery operates and is utilized and are going to be data present throughout this document and in the final product; the visual display to interface with the set of different batteries.

$$\mathrm{SOC} = \frac{\mathrm{C_{releasable}}}{\mathrm{C_{rated}}} \times 100\% \tag{1}$$

$C_{releasable}$ is the capacitance of the battery measured at any moment. C_{rated} is the stipulated capacity given by the manufacturer. SOC will never be 100% due to manufacturing itself not being perfect and 100% efficient. Also, to be taken into consideration is the lifecycle of a large capacity battery will degrade over time and, depending on the model, there’s a minimum threshold for discharging in order to maximise its duration.

$$\mathrm{SOH} = \frac{\mathrm{C_{max}}}{\mathrm{C_{rated}}} \times 100\% \tag{2}$$

C_{max} is the range at which it may operate, meaning, the total energy it yields from full charge until discharge. Overtime, SOH will diminish.

$$\mathrm{DOD} = \frac{1}{\mathrm{SOC}} \tag{3}$$

The BMS controls each aspect of SOC, SOH, temperature, voltage and current. The BMS operates as a control system for a battery and I will present some specifications of a Lithium Balance model [6].

The s-BMS (scalable) can be applied in automotive cases but also industrial and storage solutions, hence, it has a broad range of purposes and capabilities, namely communication protocol, safety and performance.

Communication protocol entails receiving and sending data in a constant and accurate way between the main controller and auxiliary equipment such as chargers, circuit breakers, switches, sensors and displays. Safety regards aspects of cell balance, voltage and current thresholds and overheating. The multiple cells of a module must be as balanced as possible so to avoid errors. As for current and voltage it is necessary to avoid short-circuits, overvoltage and also make sure the module does not overstep the established maximum temperature.

A BMS has specific current and voltage thresholds which depend on the sampling rate which, in this case, is 5Hz meaning certain current spikes that are too quick may not get detected namely when the battery is being initially charged. Figure 4 represents the safety levels and status.

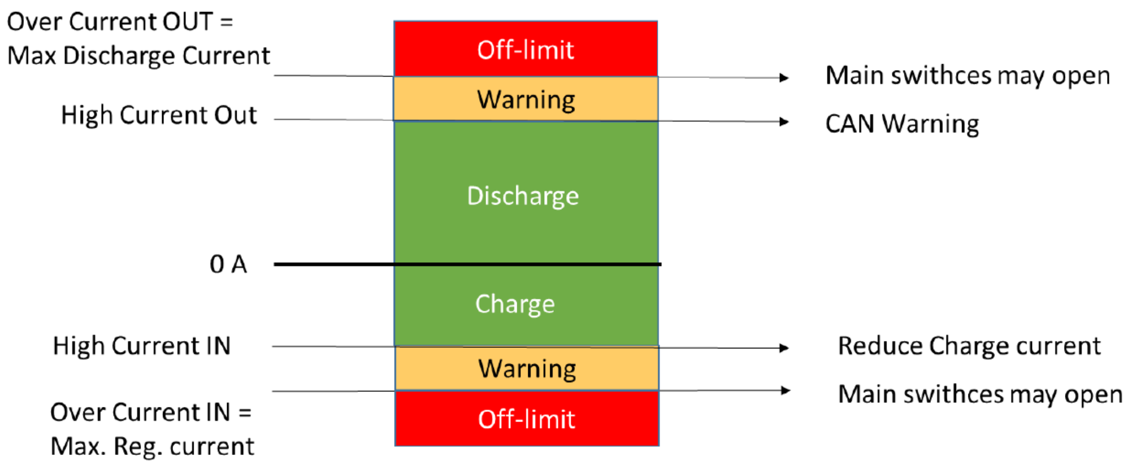


Figure 4 - Overview of safety thresholds relating to the battery current [5]

Cell balancing is a crucial part of the operation of a module and it implies that the BMS seeks the constant equilibrium of the voltage values of their cells. This can be achieved by method of bleeding which involves transmitting energy from the highest voltage cell to the lower ones in order to equilibrate them.

The objective is to limit each cell to “cell target voltage” and not to let it go lower the “minimum target voltage” and this is achieved by charging the cells at the same speed at first. Then, as different cells may have different capacities and be more or less degraded than others, they will get to 100% SOC sooner. This cell that has reached maximum charging state now will flow its voltage to the cells that are below the minimum voltage range specified and they will get charged until the s-BMS operation balances them to the same levels.

Lithium Balance, as well as other BMS systems, can provide individual cell voltages which is quite important since there might be discrepancies among them in terms of voltage and other factors. This way, the BMS alerts the user for differences in values that would otherwise remain unnoticed and could not only degrade the system but cause other unintended, hazardous consequences. So, a BMS, regardless of the specific model or battery is being used on, essentially gathers all sort of information related to the battery or set of batteries and analyses it with the main objectives of creating setpoints and alarms in case of breach of values, namely, overvoltage and temperature (most important ones). After this main, primary function, the BMS can also control and manipulate the values, to a certain extent, as far as chemical, technical and functional limitations of the battery it is embedded on go, in order to regulate them for security reasons or to improve its efficiency. In the former, it can lower voltage, temperature and cut off current if necessary. As for the latter, it aims to optimise the usage and life cycle of a battery or set of batteries without compromising the integrity and proper functioning of the device.

There is a need for specific and accurate sensors to measure temperature, balancing amongst the cells, voltage, current as well as a vast and encompassing communication protocol. To be taken into consideration as well is the cell limit that is supported for a BMS unit and the respective LMU (Local Management Unit) slave board, the latter being directly beneath the hierarchical structure of the former and having the possibility of multiple units, depending on how many modules and cells the s-BMS aims to control. The BPU (Battery Protection Unit) is a unit that guarantees the safety and operability of the entire system.

Depending on each manufacturer, there are thresholds to be considered, namely in terms of temperature, humidity, air pressure. I will present some technical specifications of Lithium Balance's s-BMS board:

Weight: BMCU 86g, LMU 72g;

Maximum number of LMUs: 32;

Maximum cells per LMU: 8;

Maximum battery capacity: 2000 Ah;

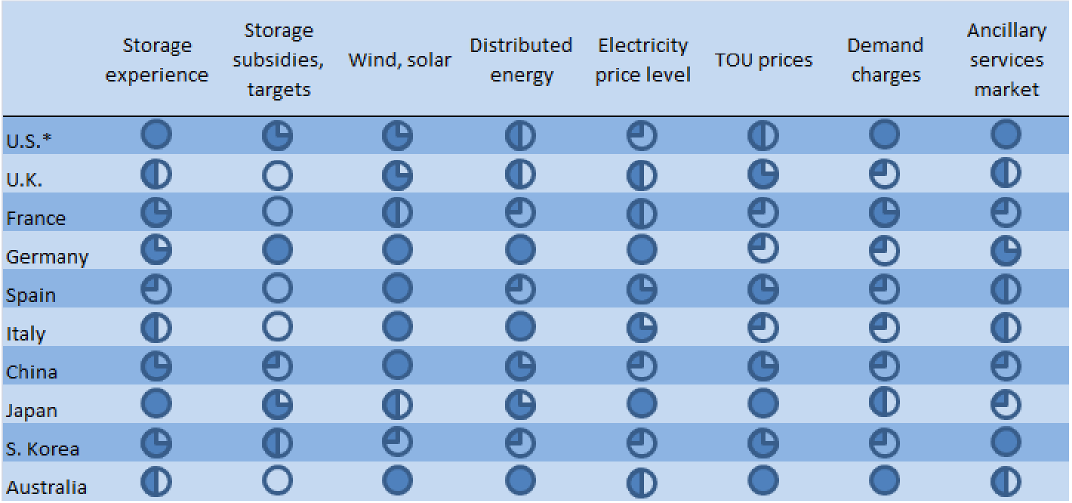
Maximum battery voltage: 1000V.

The power consumption is at around 2.1W for the BMCU (Battery Management Controller Unit) and 0.3W for each LMU meaning the main usage of power comes from the central processor which, in turn, allocates resources and redirects computing power to the individual sub-units (LMUs) and BPU for recurring safety procedures and error messages, not to mention possible need of circuit breakers and other fault mitigation techniques.

2.3 Market perception

Due to integration of more renewable energy in the electrical grid, which may very well be endemic in the upcoming years, there has also been an increase in the procurement of storage unit worldwide.

An important part of the discussion on storage systems is the development of new technologies as well as attractiveness in different markets [6].The market, being inextricably bound to weather and climate conditions as well as region specific consumption requirements, has a heterogeneous landscape throughout the world, as can be seen in the data below.



	Storage experience	Storage subsidies, targets	Wind, solar	Distributed energy	Electricity price level	TOU prices	Demand charges	Ancillary services market
U.S.*	●	◕	◕	◑	◔	◑	●	●
U.K.	◑	○	◕	◑	◑	◕	◔	◑
France	◕	○	◑	◔	◑	◔	◕	◔
Germany	◕	●	●	●	●	◔	◔	◕
Spain	◔	○	●	◔	◕	◕	◔	◑
Italy	◑	○	●	●	◕	◔	◔	◑
China	◕	◔	●	◕	◔	◕	◔	◔
Japan	●	◕	◑	◕	●	●	◑	◔
S. Korea	◕	◑	◔	◔	◔	◕	◔	●
Australia	◑	○	●	●	◑	●	●	◑

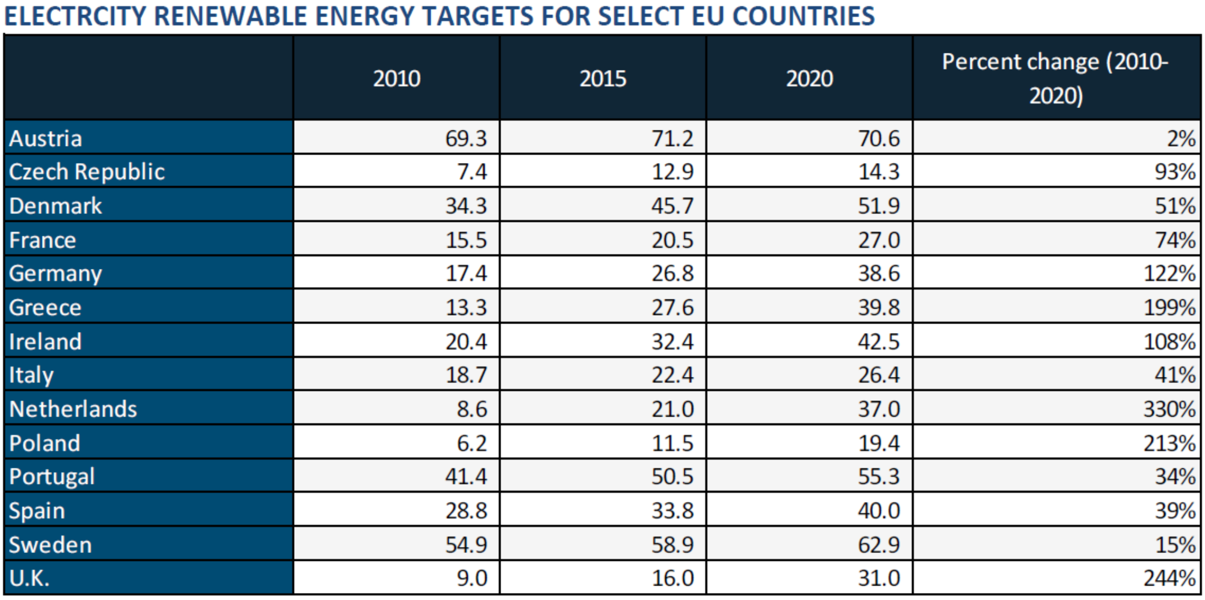
Figure 5 - Attractiveness of macro factors related to ES by region [6]

Analysing the data from Figure 5 it can be inferred there are still long strides to be made especially in demand which means this market of storage is still underdeveloped, with exceptions being the USA and Australia where there is both demand and storage experience in the former's case.

2.4 Future prospects

With the introduction of DER (Distributed Energy Resources) and respective control structures for it, the market is no longer as centralised as before, acting more and more in a mesh network. This evolution of the market implies more decentralisation and independence to the end user due to having the possibility of producing the energy it consumes as well as selling the excess to other users or universal end buyer. This will make the grid more reliable since it has several points of production, curbing single point of failures and, overall, with better quality. This interconnection is essential for the integration of renewable energy sources that, by its nature, cannot be controlled, being dependent on weather and climacteric conditions. For this, storage units have an essential role to be played by absorbing the excess energy and using it to power houses and industries when necessary.

By analysing the market data for both electrical vehicles and storage systems sales and production it can be inferred some results and predictions for this technology in the upcoming years or even decades.



ELECTRCITY RENEWABLE ENERGY TARGETS FOR SELECT EU COUNTRIES

	2010	2015	2020	Percent change (2010-2020)
Austria	69.3	71.2	70.6	2%
Czech Republic	7.4	12.9	14.3	93%
Denmark	34.3	45.7	51.9	51%
France	15.5	20.5	27.0	74%
Germany	17.4	26.8	38.6	122%
Greece	13.3	27.6	39.8	199%
Ireland	20.4	32.4	42.5	108%
Italy	18.7	22.4	26.4	41%
Netherlands	8.6	21.0	37.0	330%
Poland	6.2	11.5	19.4	213%
Portugal	41.4	50.5	55.3	34%
Spain	28.8	33.8	40.0	39%
Sweden	54.9	58.9	62.9	15%
U.K.	9.0	16.0	31.0	244%

Figure 6 - Targets of certain European countries for renewable energy penetration [6]

As can be seen form Figure 6 above, there is a significant expected increase in the injection of renewables in the grid, especially in countries like the Netherlands, U.K. and Poland, in a short period of time meaning, that this form of energy is going to become more and more common in the continent, even prevalent, in a limited amount of time.

Chapter 3

Interface development

This section will be focused on the development of an interface and its connection to the battery and respective BMS. It will show the entire development of said interface from the requisites of the prototype, to the connection to the battery and BMS as well as respective tests, building of the model and end result.

3.1 Specifications of the desired product

It will now be presented the main requirements for a working product that were accomplished throughout the development process as well as future improvements that can be made.

Main requisites:

- Provide a basic yet complete set of data upon turning on the device. This ranges from SOC, Temperature, Discharge Current, Discharge Voltage, Charge Current and Charge Voltage as the main data points that are necessary to be available throughout the entire process visualization and control process (discharge, charging…);
- Graph of SOC variation over time that can be accessed by pressing the top left button in the interface. This is important to better visualise the history of the unit in question and if it was discharged or charged and how long it took;
- Possibility of controlling the main aspects of the functioning of a storage unit.

As for future developments, the following improvements can be applied:

- Individual data of each, separate cell;
- Possibility of switching off and on cell/module as to redirect energy production or consumption as necessary;
- Others…

The interface must be, at its essence, easy to use while, at the same time, providing information and control to any user.

3.2 Architecture of the connection

The interface does not work as a separate, independent entity but, instead, is part of a more intricate system consisting of several parts that need to be operating seamlessly and in harmony with each other.

First, there is the battery, that either provides the energy to the load or receives it from a RES or electric battery charger or another source. This primary block is comprised of the BMS as well which was already described in Chapter 2 and the battery pack can have a different number of cells, depending on the model and manufacturer and modules as well.

The first block is connected to the interface by means of Controller Area Network (CANH and CANL) connections, described in more detail later on which, in turn, is connected to the visual display using the program. This will be explained further down this section.

For future development, this interface can also be connected to an inverter to the grid, namely bidirectional AC/DC. In the following diagram I will present both the work accomplished throughout the project and also possible implementation.

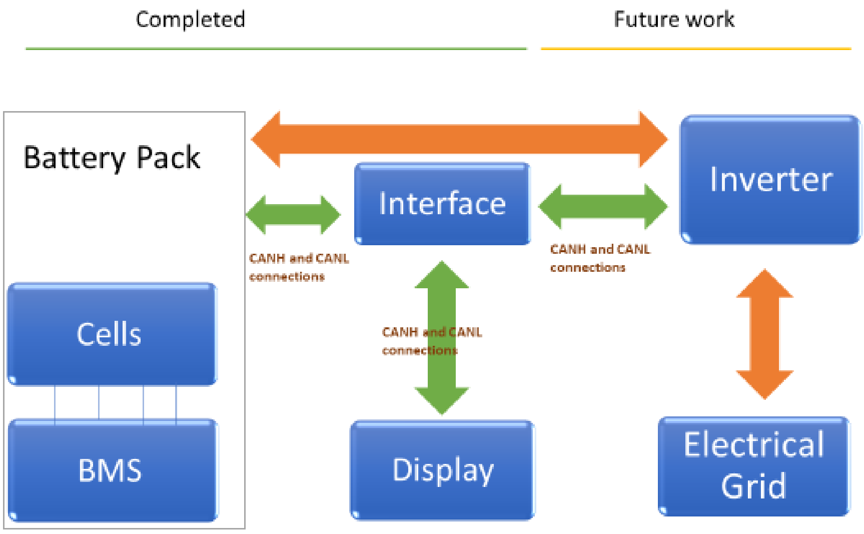


Figure 7 - Diagram of the proposed solution

The green arrows represent the data communication between the controller (BMS), battery pack, interface, display and inverter whereas the orange arrows are physical, electrical connections. The connections are all bidirectional, creating a feedback system where information is passed from the BMS to the battery, imposing limits and safety measures and the battery provides the values to be read by the BMS. The BMS also sends values to the interface and the display is a representation of the data read by the interface and, in turn, can also serve as a gateway for controlling purposes by the user.

3.3 Devices used

The following part of the project is to adapt and program an interface system using the OPUS Projektor software in conjunction with the Tesla module and SIMP BMS board. The physical display is a Wachendorff OPUSA3 which is connected from the CAN to the computer.

3.4.1 SIMP BMS Board

The board that connects the different batteries or set of batteries (as is the case of the VW modules) is custom built for this purpose and will be shown in the next figures, 8 and 9.

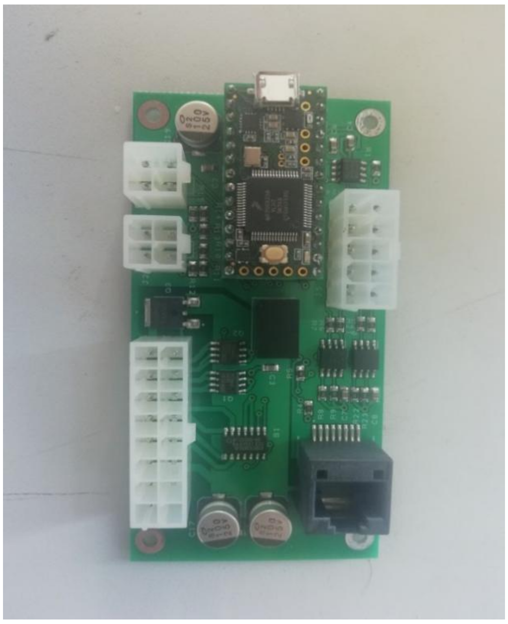


Figure 8 - SIMP BMS board (Front)

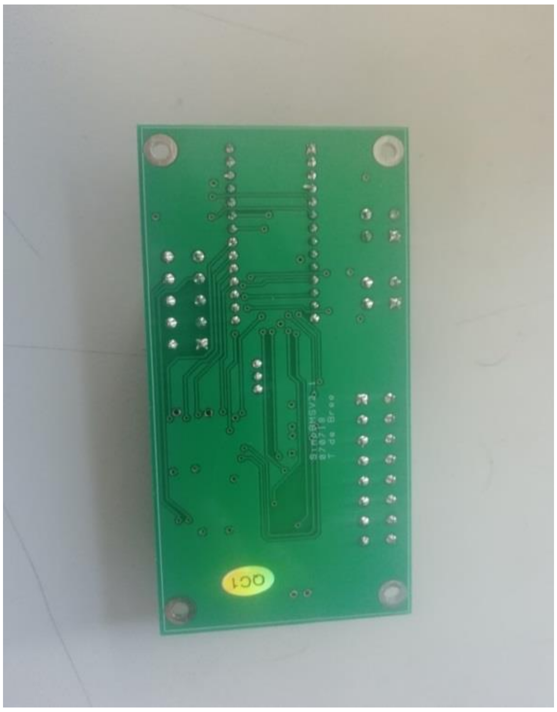


Figure 9 - SIMP BMS board (Back)

The wiring procedure and setup can be seen in Annex B and it requires a 12V source as well as CAN communication in the form of CANH (CAN High) and CANL (CAN Low) wires connected from the battery or serial port (8 connected batteries) to the board itself.

3.4.2 Physical display

The goal is to make a simple yet efficient display, shown in figures 10 and 11, that can be used by the average customer and provides all the information necessary for the best performance and usability as possible. The information is the following: SOC, SOH, Discharge Current, Discharge Voltage, Charge Current, Charge Voltage and Temperature. This information was already being read by the Arduino and board apparatus, so it was just a visual and interactive upgrade and bridging the abstraction level closer to a client/non-technical user perspective.

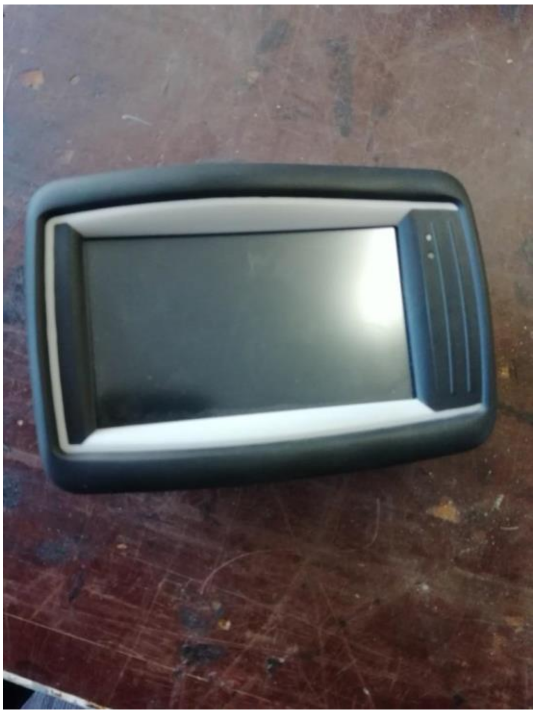


Figure 10 - Physical display (Front)

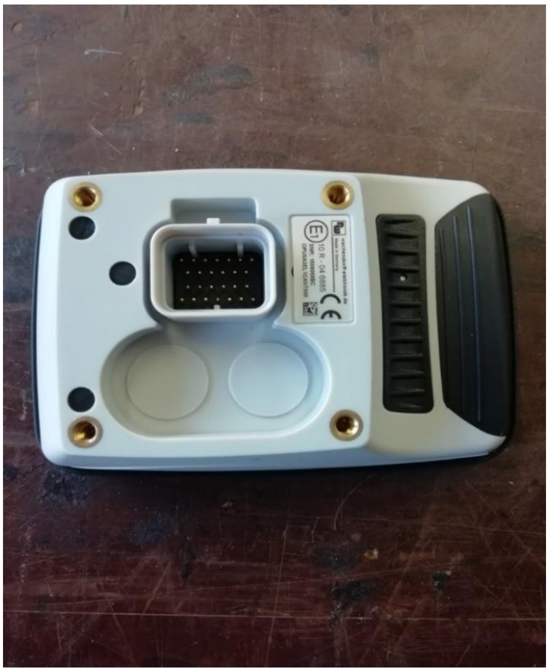
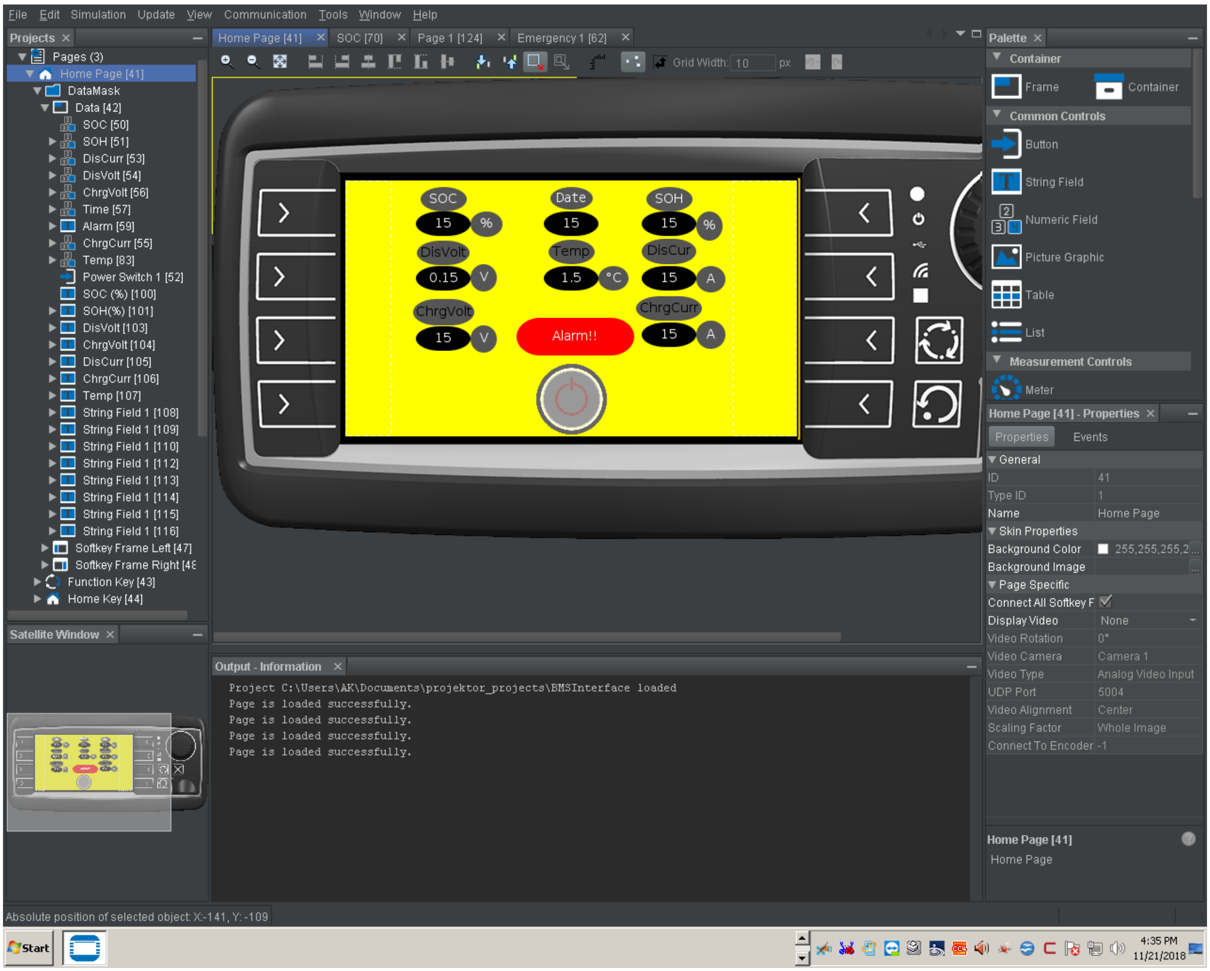


Figure 11 - Physical display (Back)

It is necessary to document the variables being used for the display, its type (integer, char or other) and where in the CAN they come from, represented in Table 1.

Table 1 - Variables in the display

Variables	Type	Value	CAN ID
chrgvolt	uint16	49100mV	0x351
chrgcurr	int	0	0x351
disvolt	uint16	42000mV	0x356
discurr	int	0	0x356
SOH	uint16	100 (Max)	0x355
SOC	int	100 (Max)	0x355
alarm	uchar	0,0,0,0	0x35A
temp	int	0	0x356
time?	int	0	
name	uchar	-	0x370
number	uchar	-	0x35E



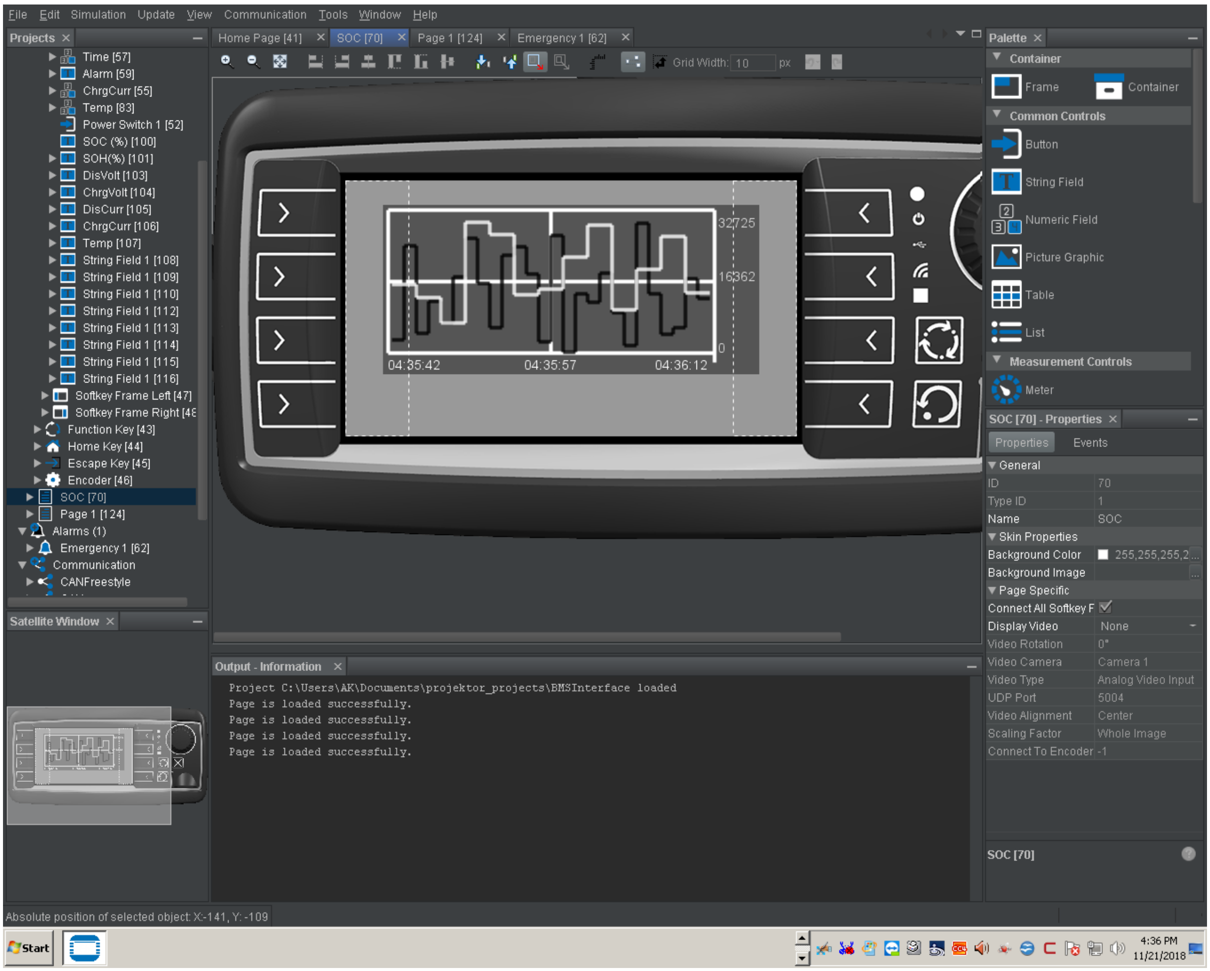
```
Project C:\Users\AK\Documents\projektor_projects\BMSInterface loaded
Page is loaded successfully.
Page is loaded successfully.
Page is loaded successfully.
Page is loaded successfully.
```

ID	41
Type ID	1
Name	Home Page
Background Color	255,255,255,2...
Background Image	
Connect All Softkey F	☑
Display Video	None
Video Rotation	0°
Video Camera	Camera 1
Video Type	Analog Video Input
UDP Port	5004
Video Alignment	Center
Scaling Factor	Whole Image
Connect To Encoder	-1

Absolute position of selected object: X:-141, Y: -109

Figure 12 - Building the interface

Some information is perceived to be of the utmost importance, especially for the end user. That is why I chose to create a graph for the SOC and how it varies throughout its usage, namely, if it was discharged the day before and how much was it used, how long it took to recharge. Temperature is also data that is relatable and easy to understand by any non-technical user however, it is linked with the alarm, in case it goes under or over the set parameters of the battery.



```
Project C:\Users\AK\Documents\projektor_projects\BMSInterface loaded
Page is loaded successfully.
Page is loaded successfully.
Page is loaded successfully.
Page is loaded successfully.
```

General	
ID	70
Type ID	1
Name	SOC
Skin Properties	
Background Color	255,255,255,2
Background Image	
Page Specific	
Connect All Softkey F	☑
Display Video	None
Video Rotation	0°
Video Camera	Camera 1
Video Type	Analog Video Input
UDP Port	5004
Video Alignment	Center
Scaling Factor	Whole Image
Connect To Encoder	-1

Absolute position of selected object: X:-141, Y: -109

Figure 13 - SOC graph editing

Using the OPUS software, I managed to design a functioning display system, seen in the previous figures, 12 and 13, where the most important part was declaring the correct IDs from Table 3 in order to get the readings from the BMS. This display works with VW and Tesla since they all have the same communication protocols, hence, it is a ubiquitous system as intended. After connecting I was able to get the values as can be seen in Figures 29 and 30, VW and Tesla respectively in the display.

Ultimately, the prime objective of said interface is, not only to visualise in a fast and seamless way the important values, when there is some sort of error, but also to control and send messages. For instance, being able to simply turn on or off each module or divert them for different loads seems to be paramount for a smart, controllable and reliable energy management system.

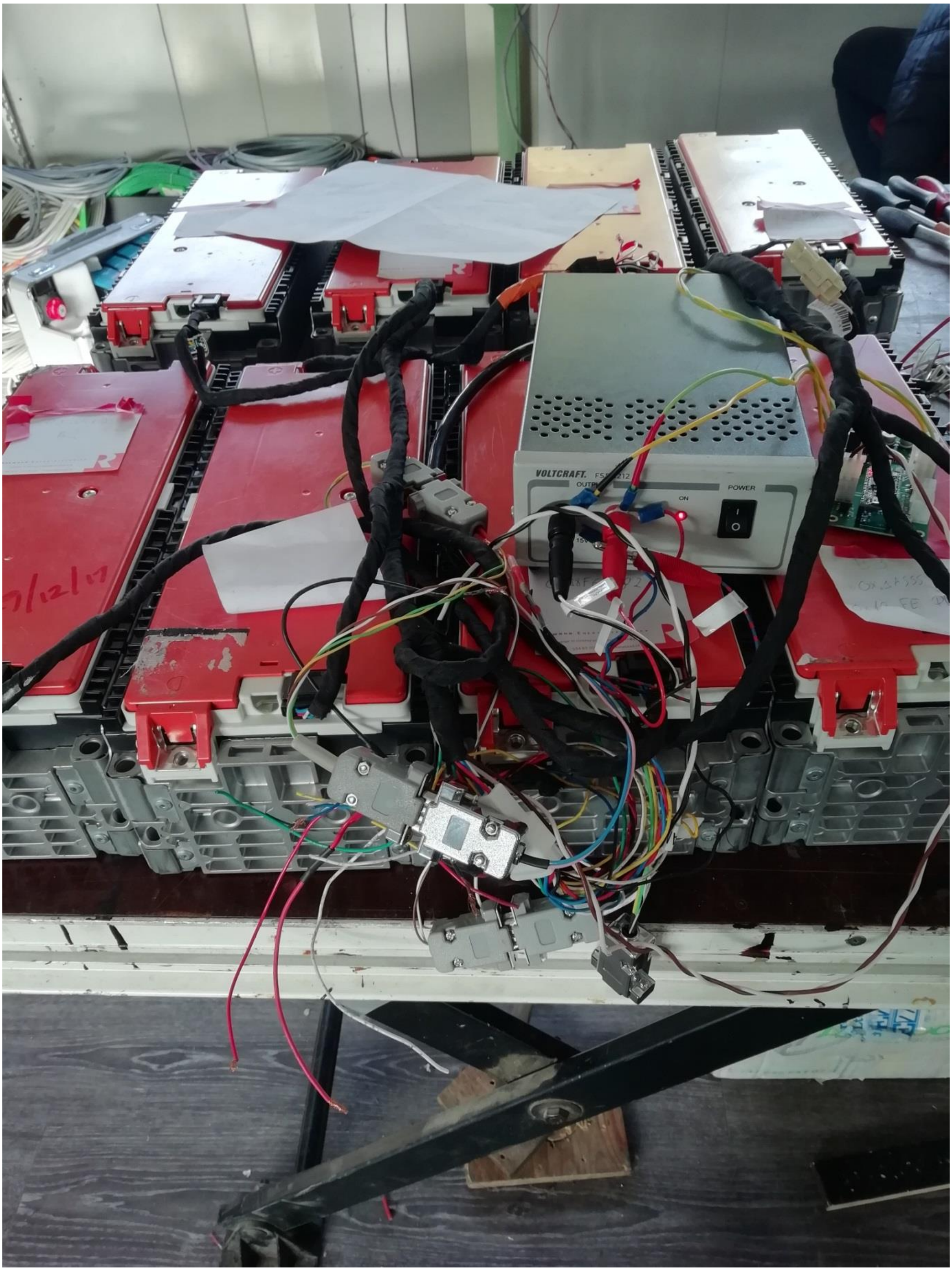


Figure 14 - Setup for the VW series

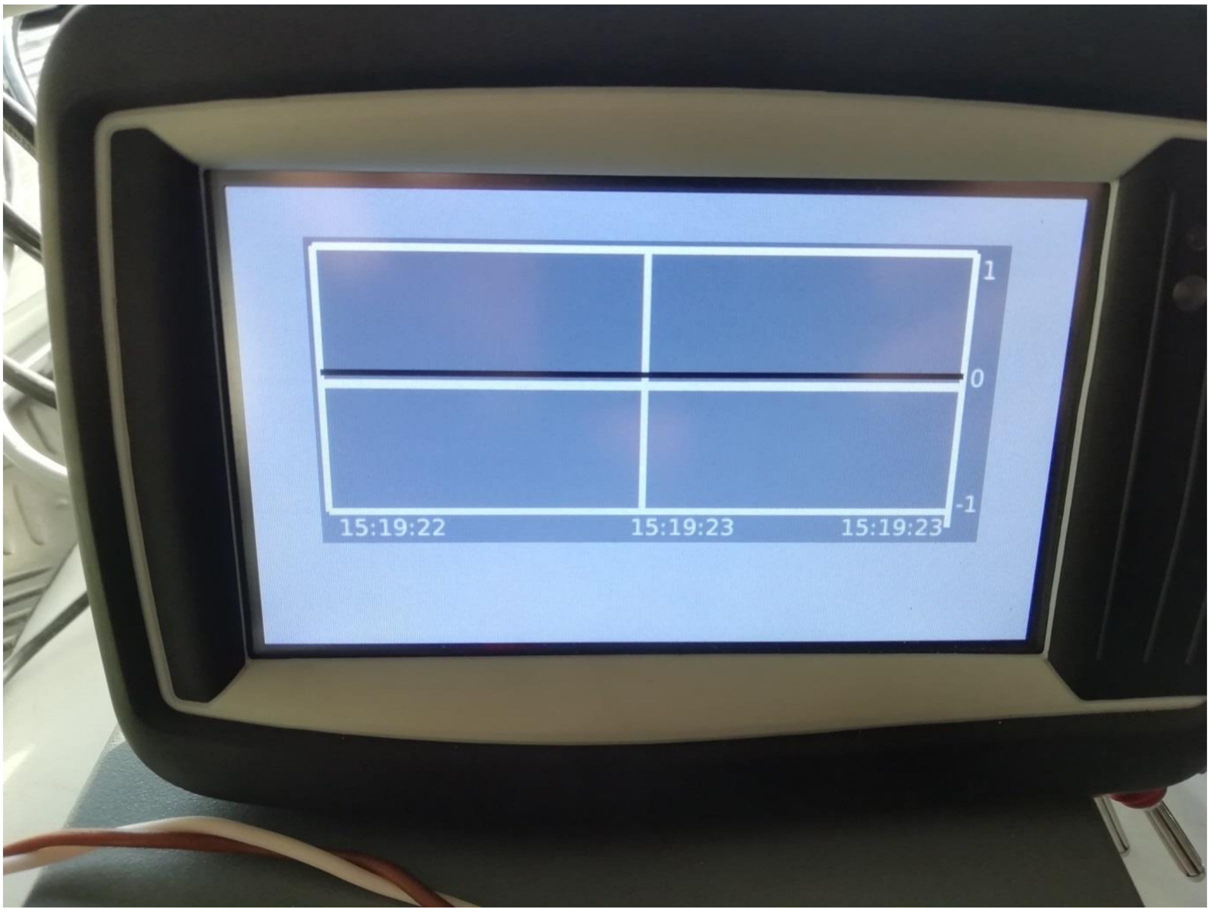


Figure 15 - SOC graph test

Figures 14 and 15 show both the complete interconnection with the 8 VW modules, 12V power source and display through use of CAN protocol and the testing of the SOC value in a graph in said physical display.

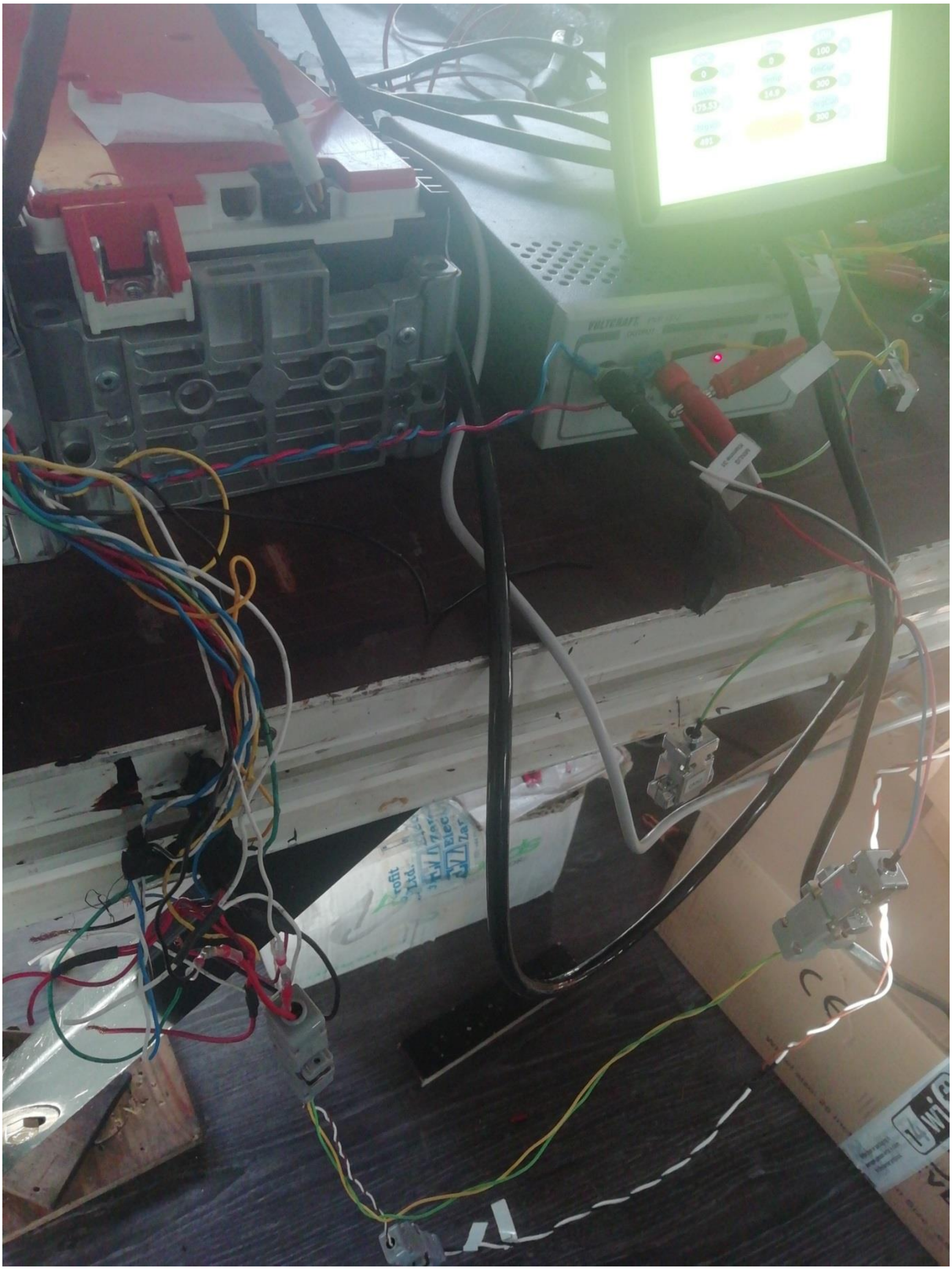


Figure 16 - Fully operational display connected to the modules

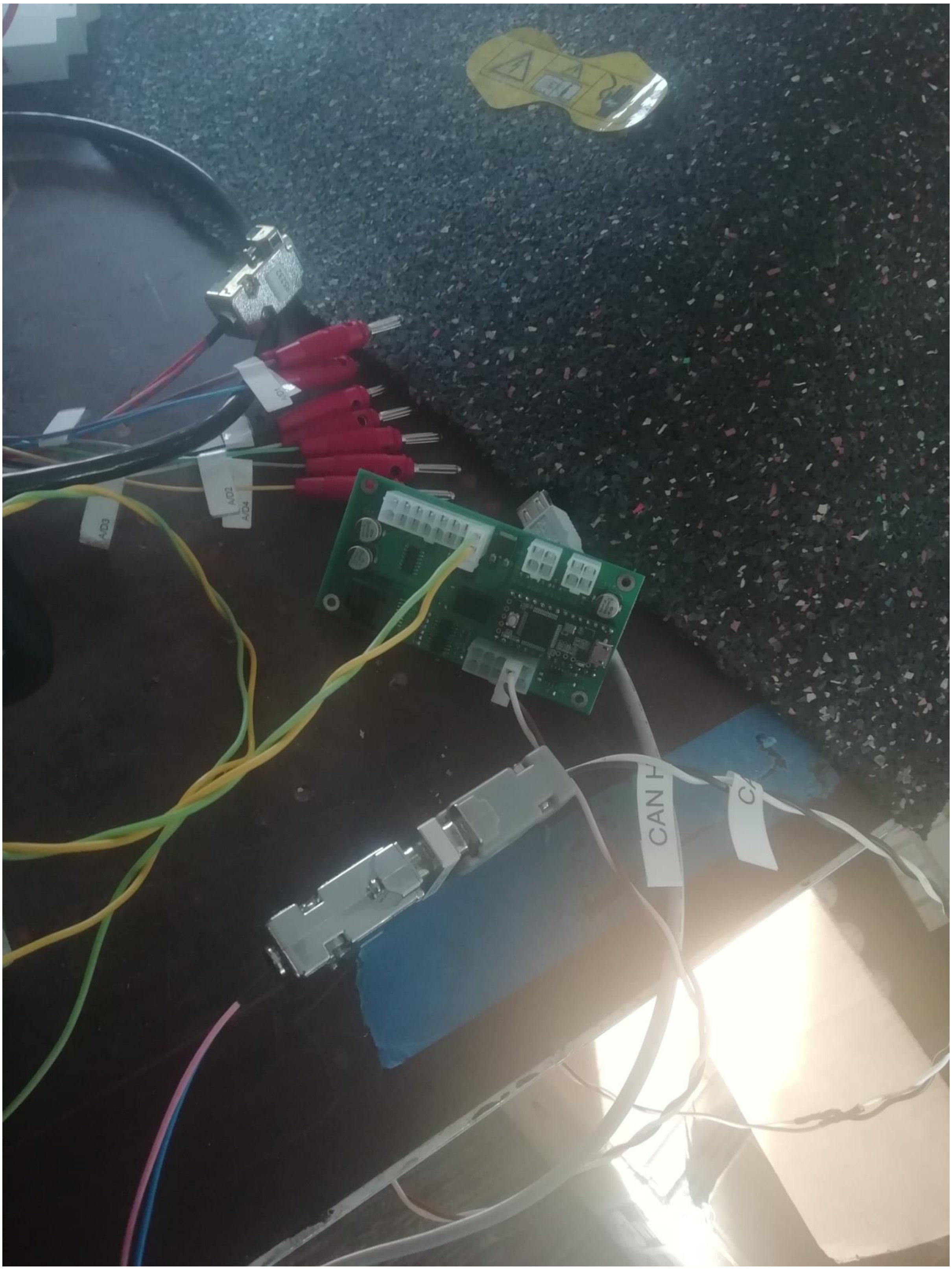
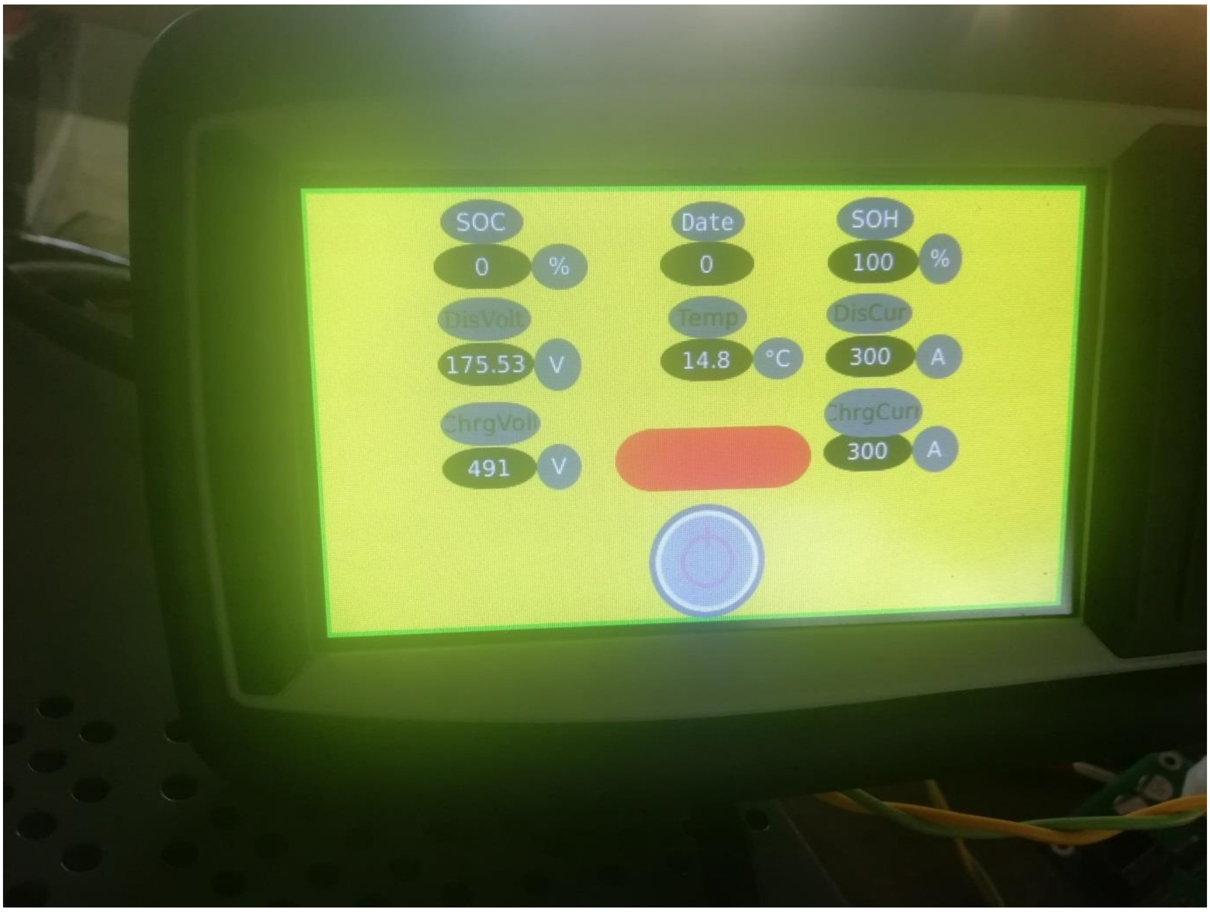


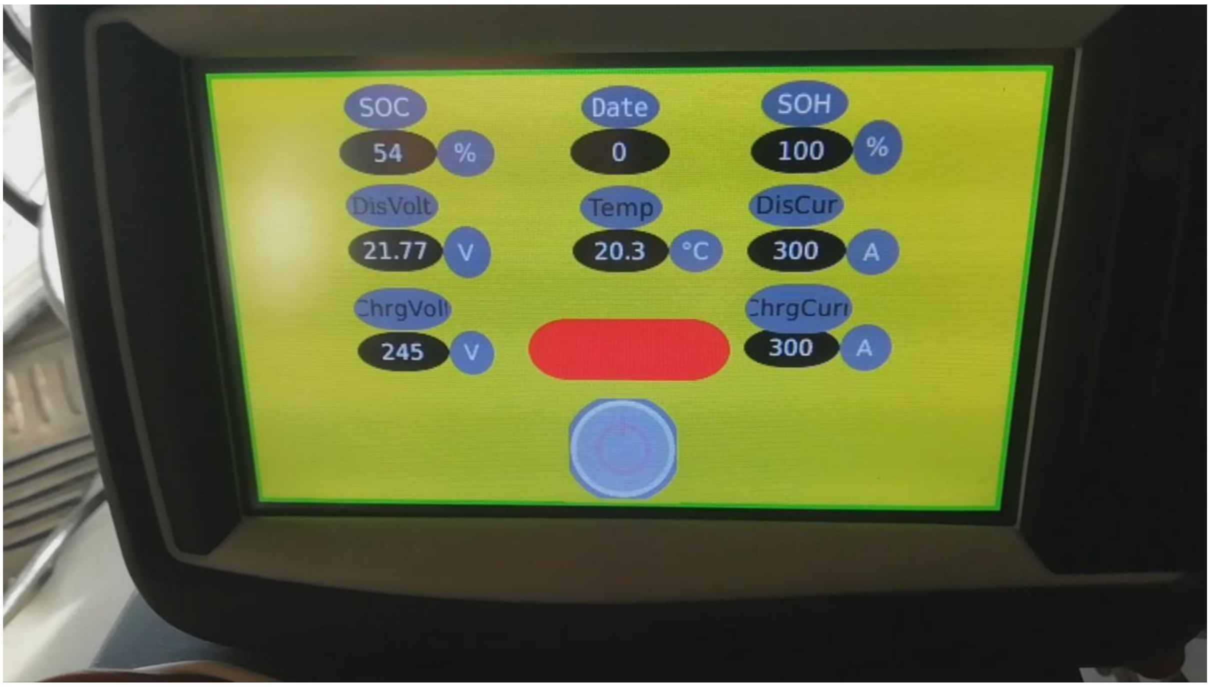
Figure 17 - Detail of the SIMP board wired to the display using CAN

Figures 16 and 17 detail the connections with the board and physical display.



SOC 0 %
Date 0
SOH 100 %
DisVolt 175.53 V
Temp 14.8 °C
DisCur 300 A
ChrgVol 491 V
ChrgCurr 300 A

Figure 18 - VW series interface



SOC 54 %
Date 0
SOH 100 %
DisVolt 21.77 V
Temp 20.3 °C
DisCur 300 A
ChrgVol 245 V
ChrgCurr 300 A

Figure 19 - Tesla interface

Figures 18 and 19 show, respectively, the values for VW (8 modules) and Tesla.

The following step is to improve the usability of the interface itself, now that it can already provide all the necessary values. This is fairly important from an adoption perspective since there is a need for simplicity and to only get the information that is necessary for a day to day usage without being too cluttered by technical data.

The main data that ought to be available in the home screen of the display is SOC, current and voltage as well as temperature. Furthermore, there is a list of options that can be implemented:

1. Being able to control how many modules are active at a certain time and the voltage they provide;
2. Switch between storage (receiving) and battery (feeding);
3. Detailed information about tariffs and total energy produced.

These options can improve the storage system efficiency.

3.4 Programming the microcontroller

The software used for Arduino Due was open source mostly from Tom de Bree and the main program can be consulted in Annex C. As far as communication is concerned, the Victron VE Can protocol, namely the addresses it communicates through, is the same for the VW and Tesla, as can be seen in Table 3. Since the communication protocol is the same, this makes the interface being built more universal due to the possibility of being used for more than one type/model of batteries, increasing its versatility and usability turning it into a more ubiquitous system overall.

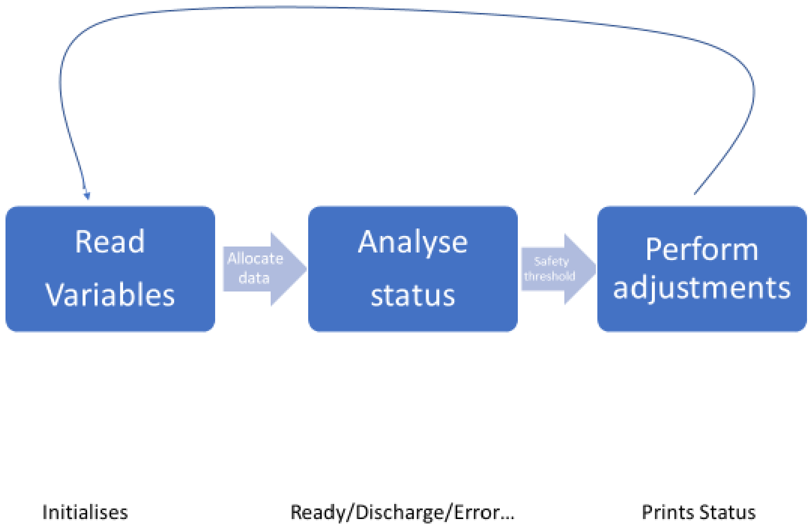
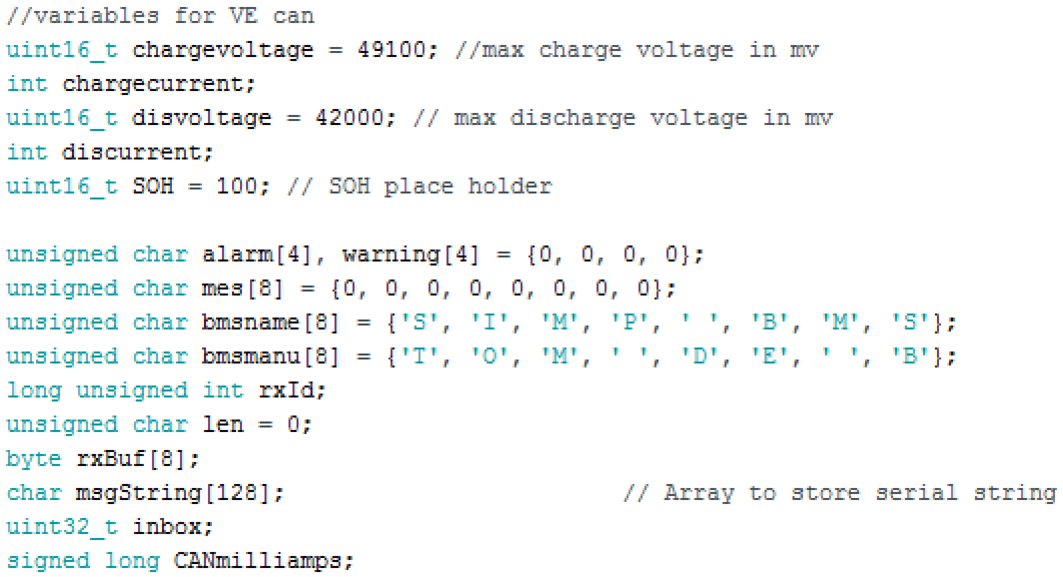


Figure 20 - Basic fluxogram of program iteration

This fluxogram (Figure 20) is but a simple representation of the most basic function of the controller to its core; the processing of data, in the form of variables such as voltage and current, yielding of status, dependent on aforementioned variables and thresholds and consequential status. It is then possible to change the values and thresholds.

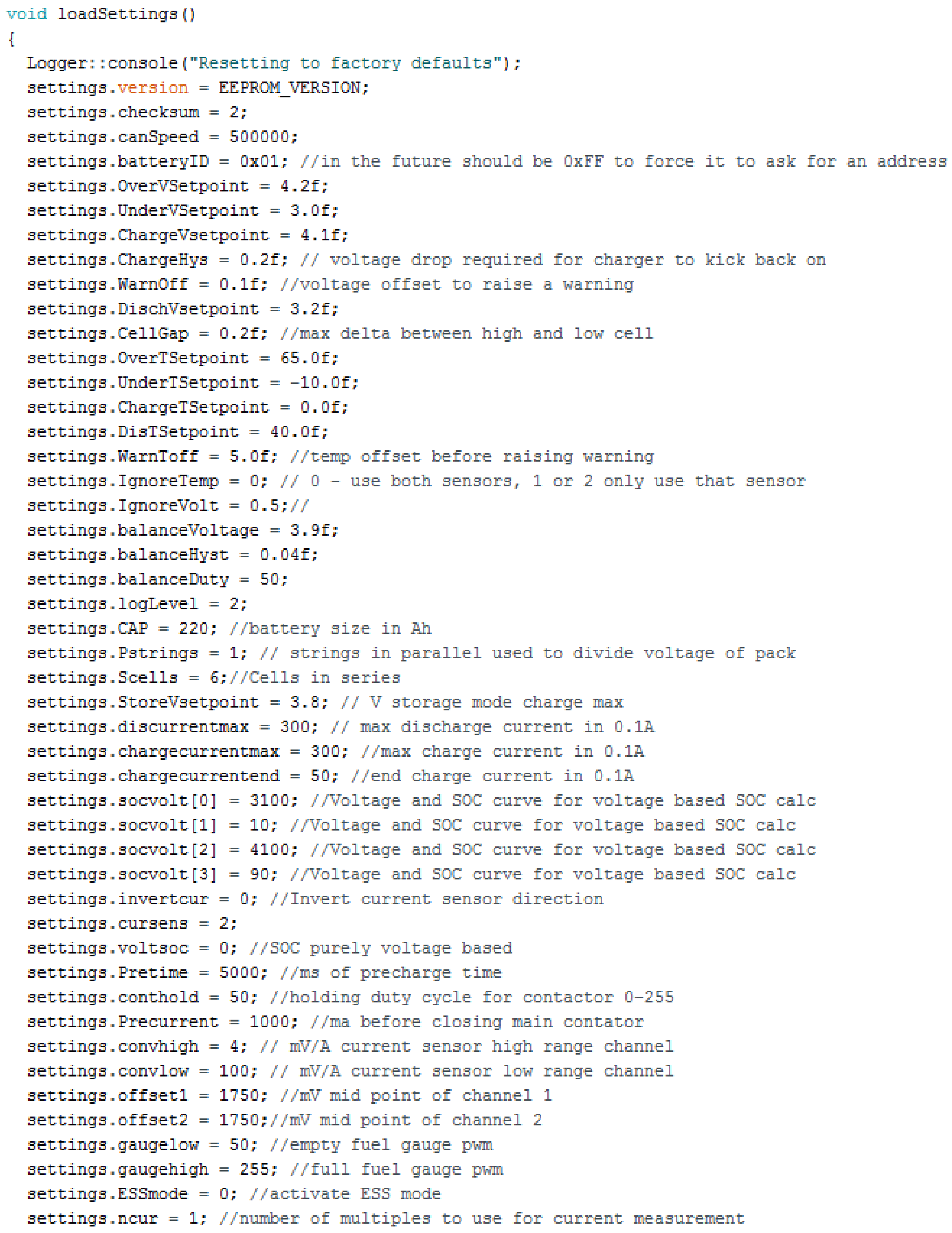


```
//variables for VE can
uint16_t chargevoltage = 49100; //max charge voltage in mv
int chargecurrent;
uint16_t disvoltage = 42000; // max discharge voltage in mv
int discurrent;
uint16_t SOH = 100; // SOH place holder

unsigned char alarm[4], warning[4] = {0, 0, 0, 0};
unsigned char mes[8] = {0, 0, 0, 0, 0, 0, 0, 0};
unsigned char bmsname[8] = {'S', 'I', 'M', 'P', ' ', 'B', 'M', 'S'};
unsigned char bmsmanu[8] = {'T', 'O', 'M', ' ', 'D', 'E', ' ', 'B'};
long unsigned int rxId;
unsigned char len = 0;
byte rxBuf[8];
char msgString[128];                        // Array to store serial string
uint32_t inbox;
signed long CANmilliamps;
```

Figure 21 - Snippet of the BMS code for VW [7]

It can be seen in the Figure 21 above some variables used for the code, namely SOH, the maximum voltage for discharge and charging processes which depends on the battery itself and it is dictated by the respective BMS.

As for the Tesla situation, the code, namely the communication protocol with the Victron VE Can as well as the settings to be specified and changed for a particular module, follows the same structure as for the VW so I choose to present the different programs interchangeably since the main function of the code presented is the same and can be analysed like that.



```
void loadSettings()
{
  Logger::console("Resetting to factory defaults");
  settings.version = EEPROM_VERSION;
  settings.checksum = 2;
  settings.canSpeed = 500000;
  settings.batteryID = 0x01; //in the future should be 0xFF to force it to ask for an address
  settings.OverVSetpoint = 4.2f;
  settings.UnderVSetpoint = 3.0f;
  settings.ChargeVsetpoint = 4.1f;
  settings.ChargeHys = 0.2f; // voltage drop required for charger to kick back on
  settings.WarnOff = 0.1f; //voltage offset to raise a warning
  settings.DischVsetpoint = 3.2f;
  settings.CellGap = 0.2f; //max delta between high and low cell
  settings.OverTSetpoint = 65.0f;
  settings.UnderTSetpoint = -10.0f;
  settings.ChargeTSetpoint = 0.0f;
  settings.DisTSetpoint = 40.0f;
  settings.WarnToff = 5.0f; //temp offset before raising warning
  settings.IgnoreTemp = 0; // 0 - use both sensors, 1 or 2 only use that sensor
  settings.IgnoreVolt = 0.5;//
  settings.balanceVoltage = 3.9f;
  settings.balanceHyst = 0.04f;
  settings.balanceDuty = 50;
  settings.logLevel = 2;
  settings.CAP = 220; //battery size in Ah
  settings.Pstrings = 1; // strings in parallel used to divide voltage of pack
  settings.Scells = 6;//Cells in series
  settings.StoreVsetpoint = 3.8; // V storage mode charge max
  settings.discurrentmax = 300; // max discharge current in 0.1A
  settings.chargecurrentmax = 300; //max charge current in 0.1A
  settings.chargecurrentend = 50; //end charge current in 0.1A
  settings.socvolt[0] = 3100; //Voltage and SOC curve for voltage based SOC calc
  settings.socvolt[1] = 10; //Voltage and SOC curve for voltage based SOC calc
  settings.socvolt[2] = 4100; //Voltage and SOC curve for voltage based SOC calc
  settings.socvolt[3] = 90; //Voltage and SOC curve for voltage based SOC calc
  settings.invertcur = 0; //Invert current sensor direction
  settings.cursens = 2;
  settings.voltsoc = 0; //SOC purely voltage based
  settings.Pretime = 5000; //ms of precharge time
  settings.conthold = 50; //holding duty cycle for contactor 0-255
  settings.Precurrent = 1000; //ma before closing main contator
  settings.convhigh = 4; // mV/A current sensor high range channel
  settings.convlow = 100; // mV/A current sensor low range channel
  settings.offset1 = 1750; //mV mid point of channel 1
  settings.offset2 = 1750;//mV mid point of channel 2
  settings.gaugelow = 50; //empty fuel gauge pwm
  settings.gaugehigh = 255; //full fuel gauge pwm
  settings.ESSmode = 0; //activate ESS mode
  settings.ncur = 1; //number of multiples to use for current measurement
```

Figure 22 - Specifications for the Tesla battery [7] with changes

Focusing on the "settings.CAP" function, seen on Figure 22, the value was changed from 100 to 220 Ah to more accurately represent the battery since it is its stated capacity. The speed that the CAN is operating at is 500k Bd and is imperative that its speed be in sync with the BMS otherwise there will be no communication possible between the entities and no information being received and transmitted. This part of the code was changed as well as some "print"s in the console for better visibility of the information being provided. The ESS mode yields the possible errors such as Overvoltage due to cell unbalancing or Undervoltage and can be used by setting it to 1.

3.5 Smart grid/metering integration

For this last step it is necessary to convert energy and transmit it to and from the grid, as well as DERs that the user may have installed in the electrical structure itself. For that, not only an accurate and complex control system is fundamental for the operation but also reliable and effective DER and RES as well as AC/DC inverter.

Since the prototype is a 50kWh storage unit meaning either 10 Tesla modules or 50 VW modules (or a combo of both) there is a need to keep them balanced and loaded so voltage, current and temperature info are important basis but phasing (Φ) is a critical element for a well-functioning grid, and that is mitigated by the AC/DC converter itself.

The infrastructure necessary for an optimised endemic storage inclusion in the daily use of an average consumer relates not only to the storage equipment itself but also ancillary services namely the regulation of certain parameters such as wave quality (frequency, flicker related issues) and the distribution of loads and control of consumption by the MGCC (Micro Grid Central Controller).

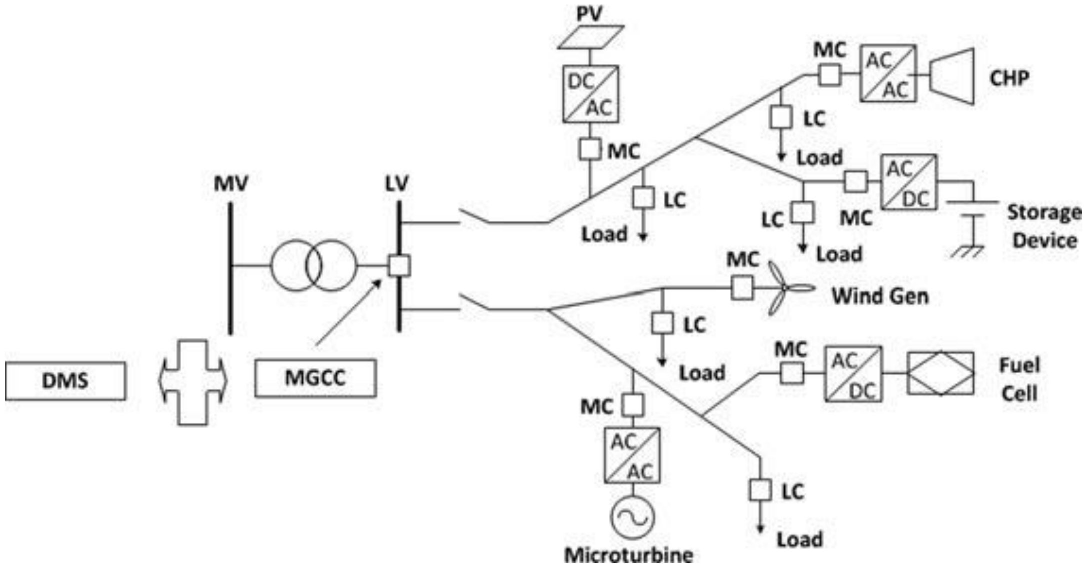


Figure 23 - Example of Microgrid Architecture [8]

In the previous structure, represented in Figure 23, it can be observed that the electric grid is powered in the Low Voltage (LV) side by RES such as wind generator and Photovoltaic (PV) cell. There can be seen a storage device and the apparatus necessary for its usage, namely, an AC/DC inverter which conducts the bidirectional interface to the grid. The storage device not only receives energy, but it can also transmit it to the grid, acting as a generator. For this, there is a MC (Microcontroller) and LC (Load Controller) upstream of the hierarchy.

The traditional infrastructure of the energy market is centralized, meaning there is a main producer of energy, usually in the form of a government monopoly company which not only builds but also oversees the entire production and transmission structure (power plants, electrical grids, control systems). The TSO (Transmission System Operator) and DSO (Distribution System Operator) act independently from each other and have the goals of streamlining the energy as best as possible without causing faults or surcharges on the grid.

The storage device, linked to the AC/DC inverter, can be the set of modules that I worked on and there is a bidirectionality involved, meaning it is both being powered, either by the grid or the RES of the facility, but it serves as battery as well, depending on the needs and desires of the user and climate/weather conditions at a certain time or even availability of the grid.

Chapter 4

Practical cases and testing

This section will go through the setup process of connecting batteries, VW in series or a single Tesla module, as well as tests of charging and discharging process, thus simulating an actual storage unit/load.

4.1 Setup

The element being studied and developed is a VW battery module which can be seen in figures 24 and 25, as well as its BMS.

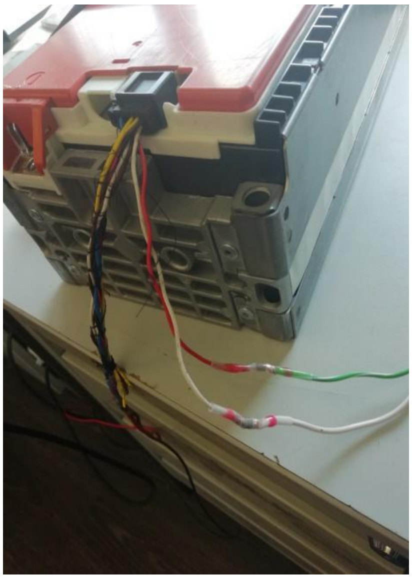


Figure 24 - VW battery module and respective CAN bus connection

These batteries have 12 cells each and have a nominal capacity of 25Ah, 1086Wh with a voltage of 44.4V per module. Each module is protected by an aluminium cover and the communication is done through an SPI interface. They have 2 internal thermistors and a low-power microcontroller. They range from 3.22V to 4.08V when they start being discharged for safety reasons.

The BMS is the device that wields information related to a battery in terms of voltage and current and their limits. It also provides information about SOC, SOH, temperature and others, depending on the specific battery and need for it. It can be used to improve and manipulate the battery and its functions as well with specific software.

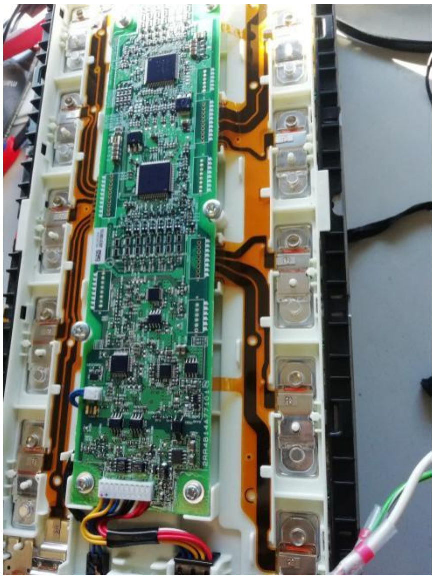


Figure 25 - VW BMS

The connection of the module with the PC and, consequently, with the software BUSMASTER is done by using a Kvaser Leaf Light HS cable (Figure 26).

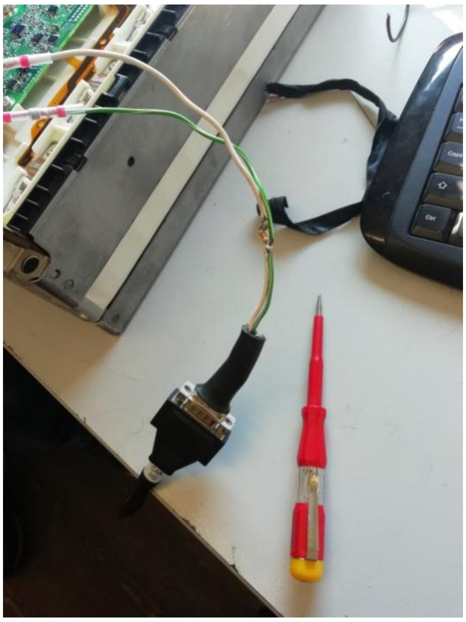


Figure 26 - Kvaser cable and connection with module

4.2 Data gathering

By using the software BUSMASTER which allows receiving CAN bus messages directly from the battery itself, I was able to collect 4 different IDs from the aforementioned module.

Table 2 - Log information from a VW battery module

Time	Direction	Channel	CAN ID	Type	DLC	Data Bytes
12:22:20:3284	Rx	1	0x1A555402	x	8	81 00 00 00 FE FE FE FE
12:22:20:3294	Rx	1	0x16A9541B	x	8	80 AA A0 AA AA A0 AA AA
12:22:20:8284	Rx	1	0x18FED202	x	5	0C 00 00 00 00
12:22:20:8294	Rx	1	0x1A555461	x	8	82 20 20 31 39 2D 00 00

Each module has specific IDs, visitble in Table 2 and provides different messages which, by previous methods of reverse engineering, it is known that the messages are related to temperature in steps of 0.5ºC for the first CAN ID and others are related to voltage and current presumably.

For each module (8 in total) the 4 discernibly IDs from the BMS can be different. Due to this uniqueness it is necessary to evaluate each battery respectively. Furthermore, after this individual analysis they will be linked together in a series. The information can be reciprocal, meaning that not only can the information be received but also sent to the BMS in a later part of the development of the project ahead.

The next step is to evaluate the entire communication protocol of the modules and respective BMSs and gather more information, represented in Table 3.

It is essential to have as many information as possible for the SOC and SOH as well due to the need of knowing the available energy for consumption and balancing with the necessity for energy at a given moment, meaning, if it meets the requirements for the need of energy. The SOH yields the battery's capacity and remaining life cycle.

Table 3 - IDs from the 8 different batteries

BUSMASTER - Exported Log File Report							
Timestamp	Direction	Channel	CAN ID	Type	DLC	Data Bytes	
00:00:00:3270	Rx	1	0x18FED302	x	5	0C 00 00 00 00	B1
00:00:00:3280	Rx	1	0x16A9541C	x	8	80 AA A0 AA AA A0 AA AA	
00:00:00:3290	Rx	1	0x1A555462	x	8	81 32 37 34 30 DB 00 00	
00:00:00:8270	Rx	1	0x1A555403	x	8	7D 00 00 00 FE FE FE FE	
00:00:00:3870	Rx	1	0x18FED402	x	5	0C 00 00 00 00	B2
00:00:00:3880	Rx	1	0x16A9541D	x	8	80 AA A0 AA AA A0 AA AA	
00:00:00:3890	Rx	1	0x1A555463	x	8	82 20 20 31 39 2D 00 00	
00:00:00:8870	Rx	1	0x1A555404	x	8	7E 00 00 00 FE FE FE FE	
00:00:00:3030	Rx	1	0x1A555402	x	8	80 00 00 00 FE FE FE FE	B3
00:00:00:3040	Rx	1	0x16A9541B	x	8	80 AA A0 AA AA A0 AA AA	
00:00:00:8030	Rx	1	0x18FED202	x	5	0C 00 00 00 00	
00:00:00:8040	Rx	1	0x1A555461	x	8	81 32 37 34 30 DB 00 00	
00:00:00:4810	Rx	1	0x18FED702	x	5	0C 00 00 00 00	B4
00:00:00:4830	Rx	1	0x16A95420	x	8	80 AA A0 AA AA A0 AA AA	
00:00:00:4830	Rx	1	0x1A555466	x	8	82 20 20 31 39 97 00 00	
00:00:00:9810	Rx	1	0x1A555407	x	8	7E 00 00 00 FE FE FE FE	
00:00:00:1970	Rx	1	0x18FED102	x	5	0C 00 00 00 00	B5
00:00:00:1980	Rx	1	0x16A9541A	x	8	80 AA A0 AA AA A0 AA AA	
00:00:00:1980	Rx	1	0x1A555460	x	8	81 32 37 34 30 DB 00 00	
00:00:00:6970	Rx	1	0x1A555401	x	8	77 00 00 00 FE FE FE FE	
00:00:00:0370	Rx	1	0x18FED802	x	5	0C 00 00 00 00	B6
00:00:00:0380	Rx	1	0x16A95421	x	8	80 AA A0 AA AA A0 AA AA	
00:00:00:0390	Rx	1	0x1A555467	x	8	82 20 20 31 39 2D 00 00	
00:00:00:5370	Rx	1	0x1A555408	x	8	7D 00 00 00 FE FE FE FE	
00:00:00:4960	Rx	1	0x18FED502	x	5	0C 00 00 00 00	B7
00:00:00:4970	Rx	1	0x16A9541E	x	8	80 AA A0 AA AA A0 AA AA	
00:00:00:4980	Rx	1	0x1A555464	x	8	80 34 31 48 20 16 00 00	
00:00:00:7960	Rx	1	0x1A555405	x	8	76 00 00 00 FE FE FE FE	
00:00:00:3910	Rx	1	0x1A555406	x	8	77 00 00 00 FE FE FE FE	B8
00:00:00:3920	Rx	1	0x16A9541F	x	8	80 AA A0 AA AA A0 AA AA	
00:00:00:8910	Rx	1	0x18FED602	x	5	0C 00 00 00 00	
00:00:00:8920	Rx	1	0x1A555465	x	8	81 31 38 34 30 EC 00 00	

Each of the 4 different ID types yields a different piece of information related to the battery:

- 0x18FEDn02 is a simple unique tag for a battery with n being the number in the series and the data it provides is irrelevant;
- 0x16A9541* is for cell balancing (which will be explored further down the document) and the data is important, A being for a balanced cell and 0 for inactive one meaning it should have as many “A”s as possible;
- 0x1A55540n (with n being module number), gives the temperature readings in 0.5 °C steps and I infer the data being FE as normal temperature readings without crossing the limits;
- 0x1A55546n seems to be an internal communication handshake between possibly different modules in series with the data having random numbers.

It was not possible to deduce and understand every single aspect of the communication protocol, namely the meaning of the transmitted “Data Bytes”, due to lack of information available for public consultation at the time by the manufacturer of the equipment. So it was purely by testing and deducing by reverse engineering that was possible to get the significance of said values.

4.3 Interconnected batteries

The following step was to connect all batteries in series through an 8 port CANbus serial cable as can be seen in figure 27.

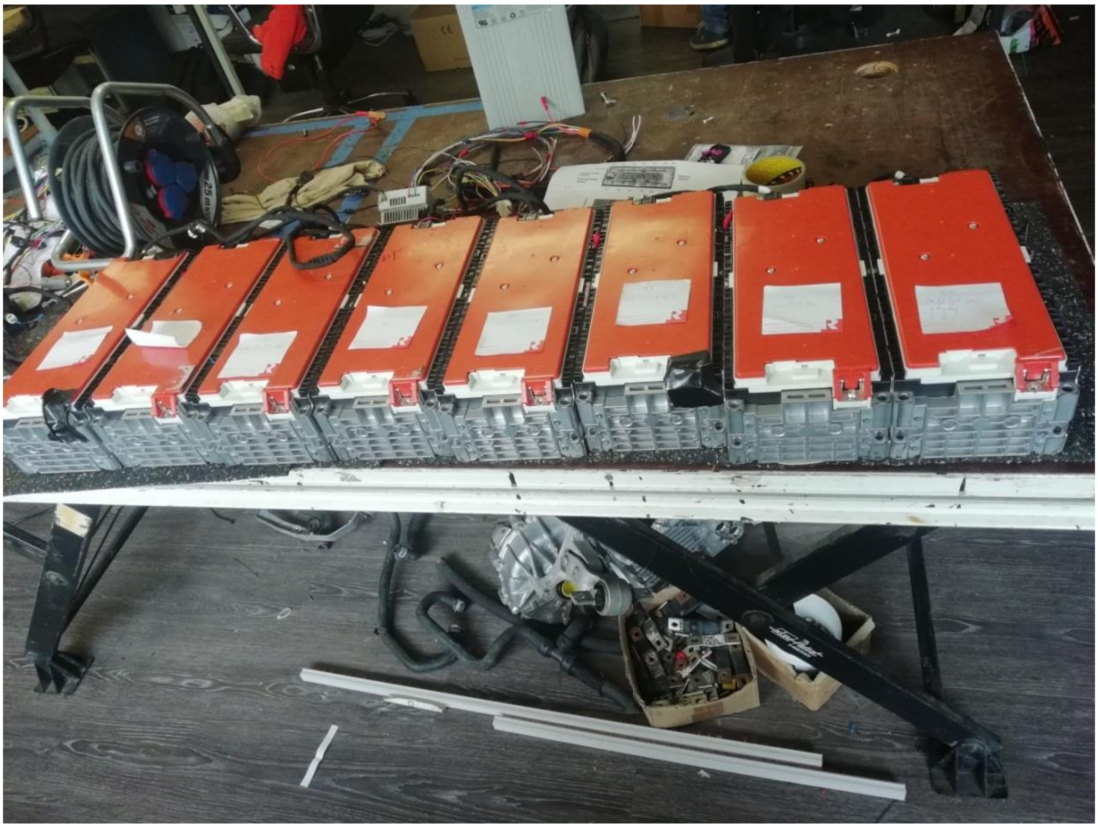


Figure 27 - Series of 8 batteries (different IDs)

The wiring diagram is included in Annex A and the schematic is shown in the next page, Figure 28, and it consists of interconnecting the CANH and CANL respectively from the battery or set of batteries in this case, to the SIMP BMS board. The batteries need to be powered by a 12V voltage source and the connection on the CAN terminals (CANH and CANL) needs to have a 60Ω resistance between them.

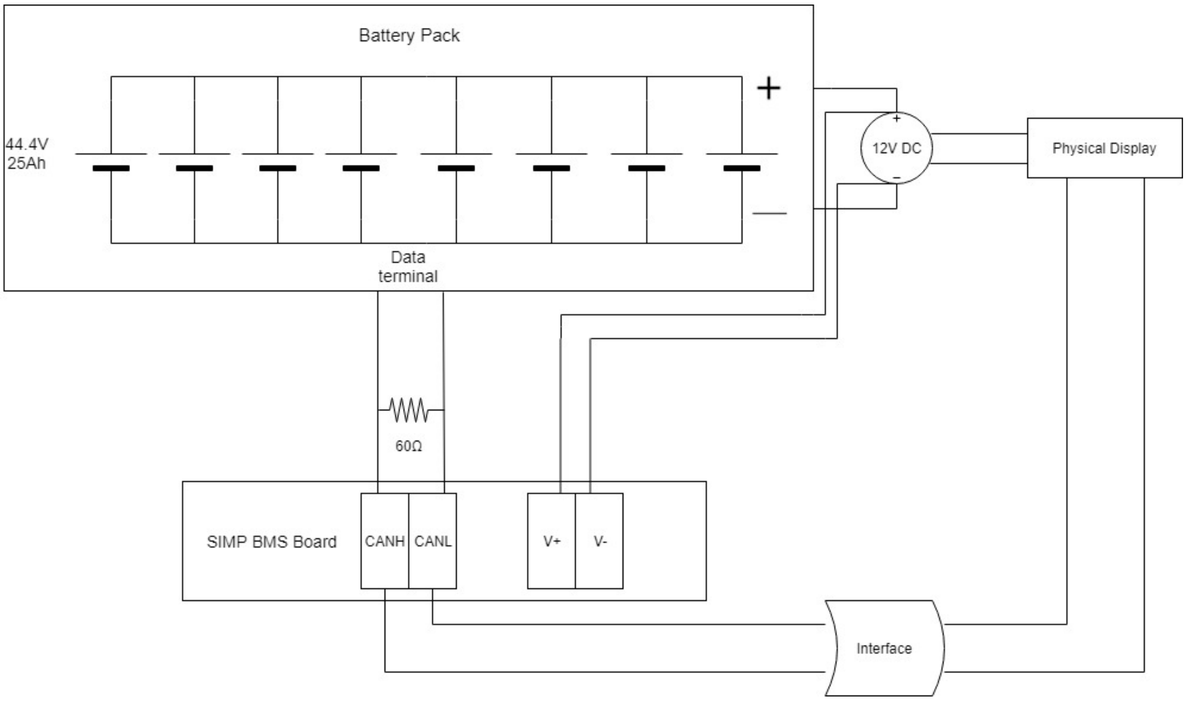


Figure 28 - Schematic of the VW series connection

The Battery Pack can be comprised of any number of modules with different specifications as long as the data communication can occur.

4.4 Module testing

In order to evaluate the efficiency and quality of a particular module it is necessary to connect it to a load (Figure 30). An electronic load was used to simulate a real-life use case; a DC Electronic Load model AT8612. The battery chosen was the B6.

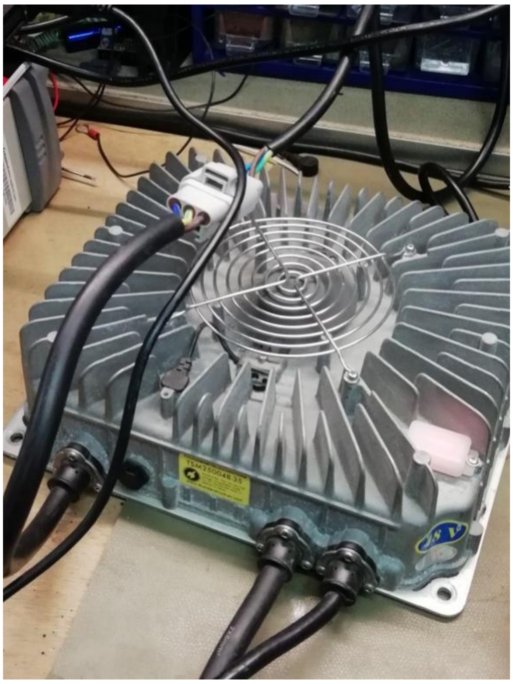
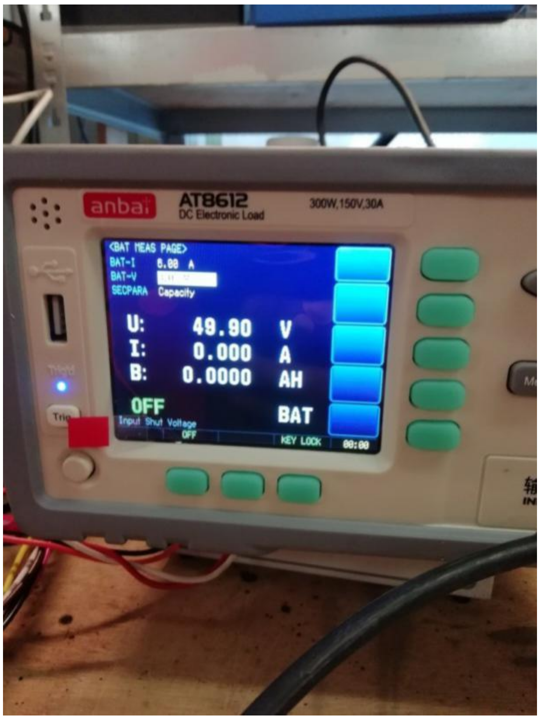


Figure 29 - Electrical battery charger

First part of the process involved charging the module with a 48V battery charger (Figure 29) connected to the electrical grid. This procedure took about 20 minutes (note that the battery, being second life, has a lower capacity than the original stated value). The battery is from July 24th 2015 meaning it was over 3 years old at the time of the testing. However, there is no way of knowing how much usage it had since it is from a recall.

Figure 30 - Electronic load AT8612

After charging the battery, I proceeded to test its capacity by connecting it to the electronic load. The threshold of the battery in terms of current and voltage is 6A and 48.9V meaning a low speed discharge, represented in Figure 31.



```
<BAT MEAS PAGE>
BAT-I     6.00 A
BAT-V
SECPARA   Capacity

U:   49.90   V
I:   0.000   A
B:   0.0000  AH

OFF                BAT
Input Shut Voltage
      OFF          KEY LOCK   00:00
```

Figure 31 - Initial values

As expected, the discharge process took significantly more time, around 3 hours to finish, without breaching the parameters of 40V and 6A since it would damage the battery to go any lower.

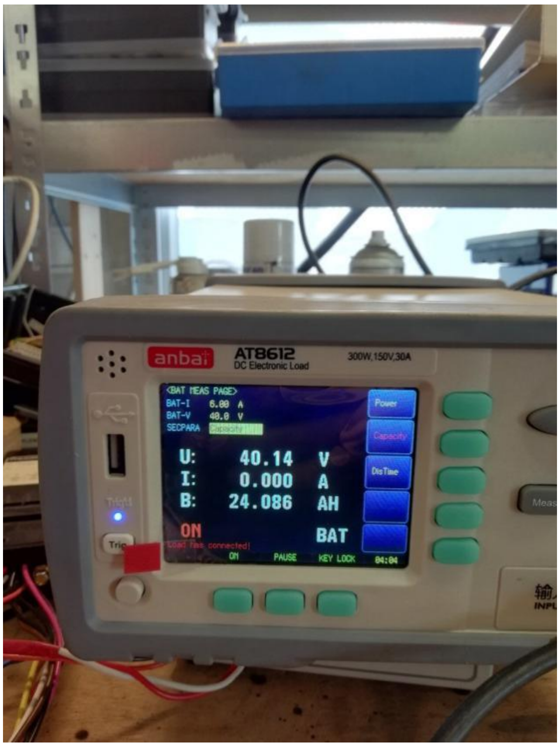


Figure 32 - Values after discharge

The value of 24Ah in a 25Ah battery represents a 96% SOC. To be taken into consideration that the battery could not lower the 40V mark, proved by Figure 32, hence it is not known if it could have powered the load even more.

4.5 Tesla module analysis

This procedure involved wiring the exit port of the CANbus communication of a Tesla module as can be seen in Figure 33.

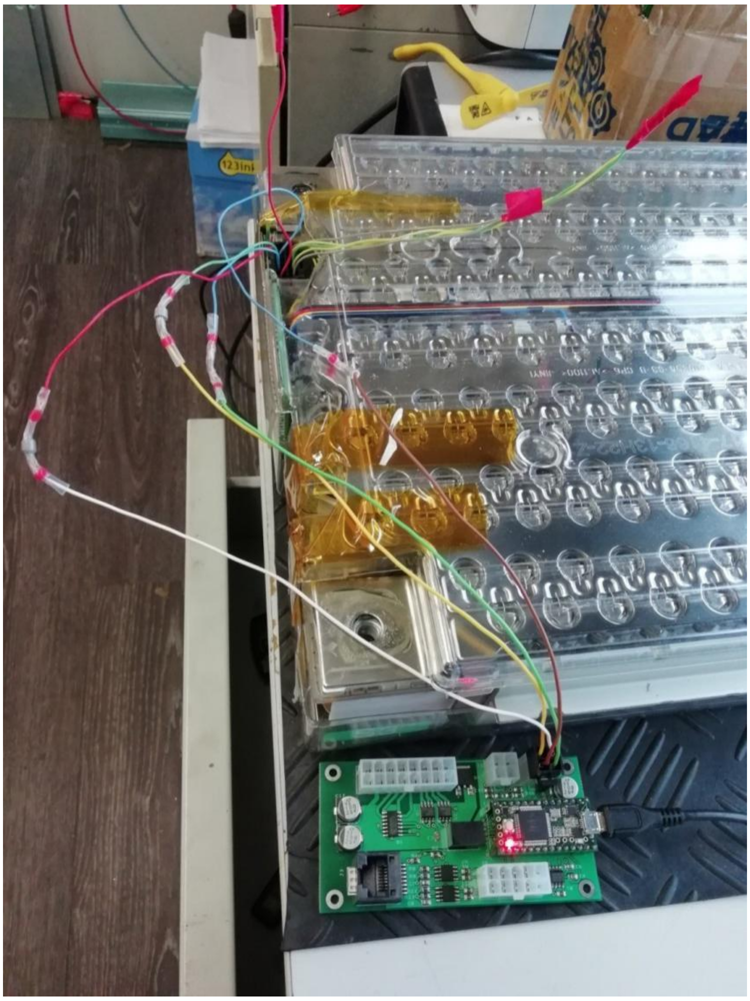


Figure 33 - Tesla module connected to SIMP BMS board

This particular module has a capacity of 232Ah, energy of 5.3KWh and the cells have a nominal voltage of 3.8V and the cutoff process occurs at 4.2V for charging and 3.3V for discharging as was proven in testing. As for the discharge current, it reaches a maximum of 750A. The cells (444 per module) have 3400mAh of capacity.

Unlike the VW model, the SIMP BMS board does not require outside powering, being powered only by the 5V output from the battery itself. The BMS is embedded in the battery and through the plastic cover, seen in Figure 34 and 35.

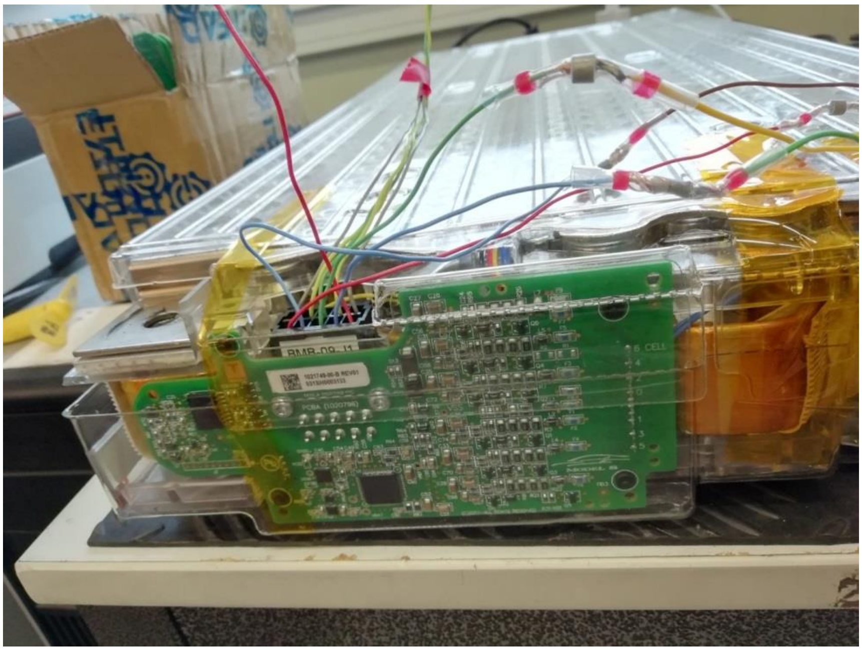


Figure 34 - Tesla BMS embedded in the battery

Using the software from Tom de Bree for Tesla and adapting it, it was possible to get data once more. This module in particular had 2 temperature sensors with 3 significant digits each, meaning it was more precise. The same can be said of the voltage that it is given, making it a more accurate estimation.

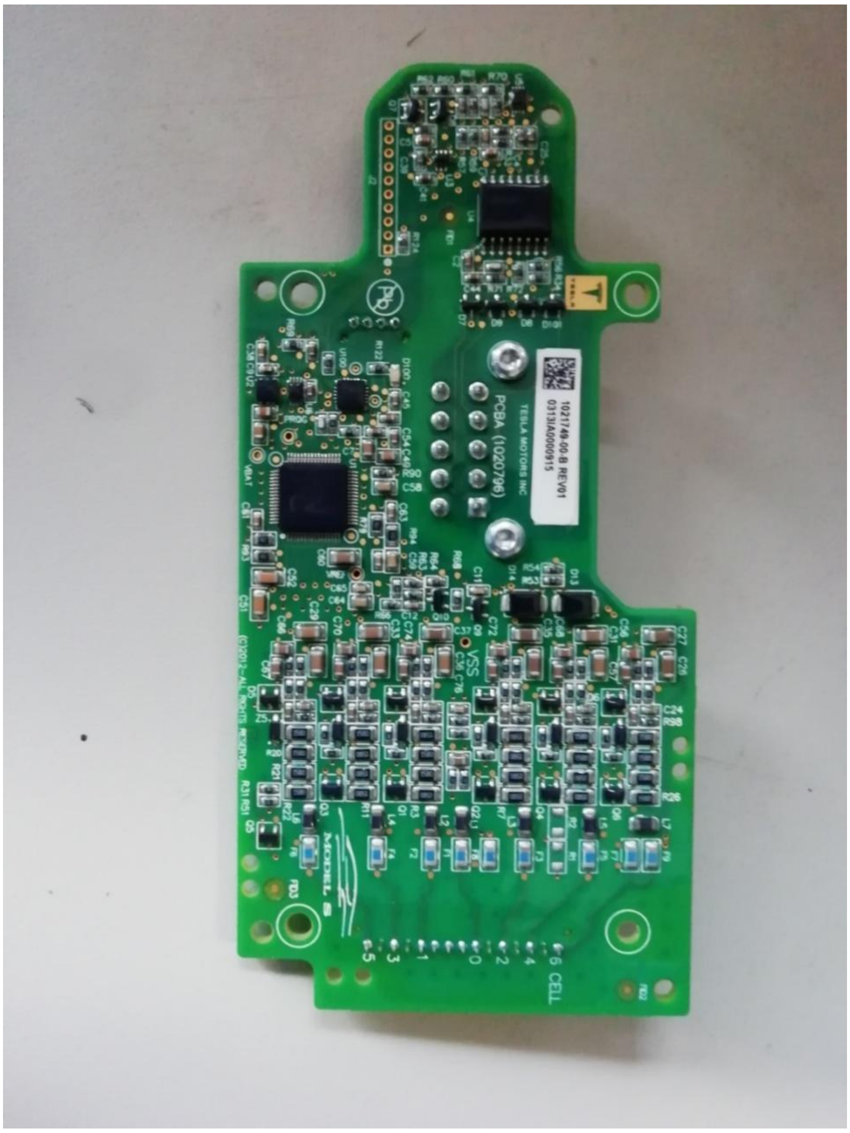
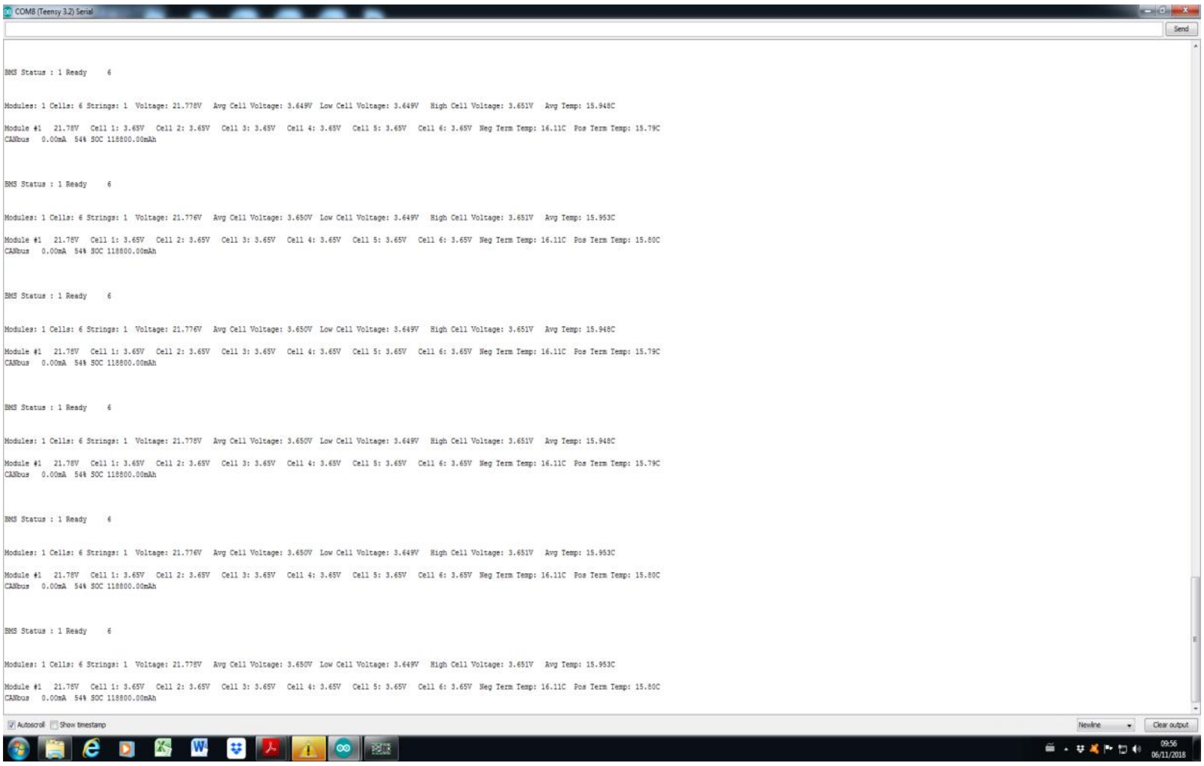


Figure 35 - Tesla's Model S BMS



```
BMS Status : 1 Ready     6


Modules: 1 Cells: 6 Strings: 1  Voltage: 21.778V   Avg Cell Voltage: 3.649V  Low Cell Voltage: 3.649V   High Cell Voltage: 3.651V   Avg Temp: 15.948C

Module #1   21.78V   Cell 1: 3.65V   Cell 2: 3.65V   Cell 3: 3.65V   Cell 4: 3.65V   Cell 5: 3.65V   Cell 6: 3.65V  Neg Term Temp: 16.11C  Pos Term Temp: 15.79C
CANbus   0.00mA  54% SOC 118800.00mAh



BMS Status : 1 Ready     6


Modules: 1 Cells: 6 Strings: 1  Voltage: 21.776V   Avg Cell Voltage: 3.650V  Low Cell Voltage: 3.649V   High Cell Voltage: 3.651V   Avg Temp: 15.953C

Module #1   21.78V   Cell 1: 3.65V   Cell 2: 3.65V   Cell 3: 3.65V   Cell 4: 3.65V   Cell 5: 3.65V   Cell 6: 3.65V  Neg Term Temp: 16.11C  Pos Term Temp: 15.80C
CANbus   0.00mA  54% SOC 118800.00mAh



BMS Status : 1 Ready     6


Modules: 1 Cells: 6 Strings: 1  Voltage: 21.776V   Avg Cell Voltage: 3.650V  Low Cell Voltage: 3.649V   High Cell Voltage: 3.651V   Avg Temp: 15.948C

Module #1   21.78V   Cell 1: 3.65V   Cell 2: 3.65V   Cell 3: 3.65V   Cell 4: 3.65V   Cell 5: 3.65V   Cell 6: 3.65V  Neg Term Temp: 16.11C  Pos Term Temp: 15.79C
CANbus   0.00mA  54% SOC 118800.00mAh



BMS Status : 1 Ready     6


Modules: 1 Cells: 6 Strings: 1  Voltage: 21.778V   Avg Cell Voltage: 3.650V  Low Cell Voltage: 3.649V   High Cell Voltage: 3.651V   Avg Temp: 15.948C

Module #1   21.78V   Cell 1: 3.65V   Cell 2: 3.65V   Cell 3: 3.65V   Cell 4: 3.65V   Cell 5: 3.65V   Cell 6: 3.65V  Neg Term Temp: 16.11C  Pos Term Temp: 15.79C
CANbus   0.00mA  54% SOC 118800.00mAh



BMS Status : 1 Ready     6


Modules: 1 Cells: 6 Strings: 1  Voltage: 21.776V   Avg Cell Voltage: 3.650V  Low Cell Voltage: 3.649V   High Cell Voltage: 3.651V   Avg Temp: 15.953C

Module #1   21.78V   Cell 1: 3.65V   Cell 2: 3.65V   Cell 3: 3.65V   Cell 4: 3.65V   Cell 5: 3.65V   Cell 6: 3.65V  Neg Term Temp: 16.11C  Pos Term Temp: 15.80C
CANbus   0.00mA  54% SOC 118800.00mAh



BMS Status : 1 Ready     6


Modules: 1 Cells: 6 Strings: 1  Voltage: 21.778V   Avg Cell Voltage: 3.650V  Low Cell Voltage: 3.649V   High Cell Voltage: 3.651V   Avg Temp: 15.953C

Module #1   21.78V   Cell 1: 3.65V   Cell 2: 3.65V   Cell 3: 3.65V   Cell 4: 3.65V   Cell 5: 3.65V   Cell 6: 3.65V  Neg Term Temp: 16.11C  Pos Term Temp: 15.80C
CANbus   0.00mA  54% SOC 118800.00mAh
```

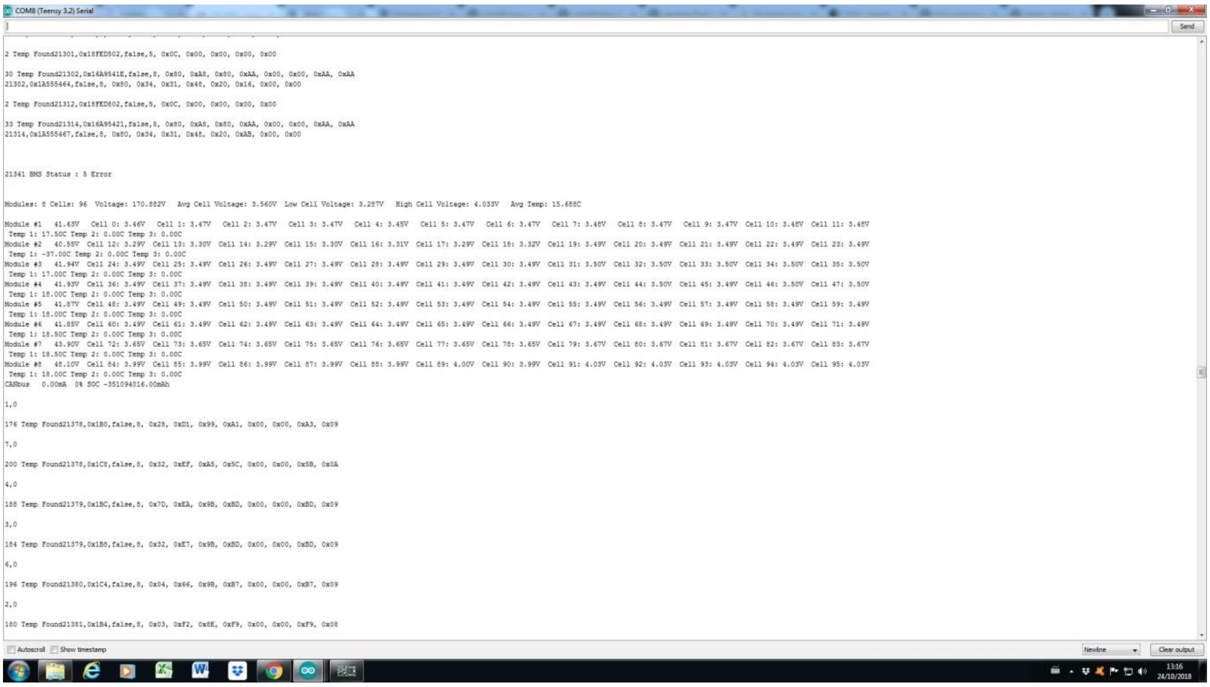
Figure 36 - Tesla menu

This menu from Figure 36 yields data from the Tesla battery which, in turn, is provided by the BMS system, seen in the previous figure. Different status such as, Ready or Error, will depend on the thresholds defined and the values at a certain time that the battery is operating at.

4.6 Feasibility

After connecting all batteries, it is possible to infer that the system of the BMS, if there's a certain imbalance between the cells of the module, does not yield the correct temperature, switching constantly to a negative outlier value by default.

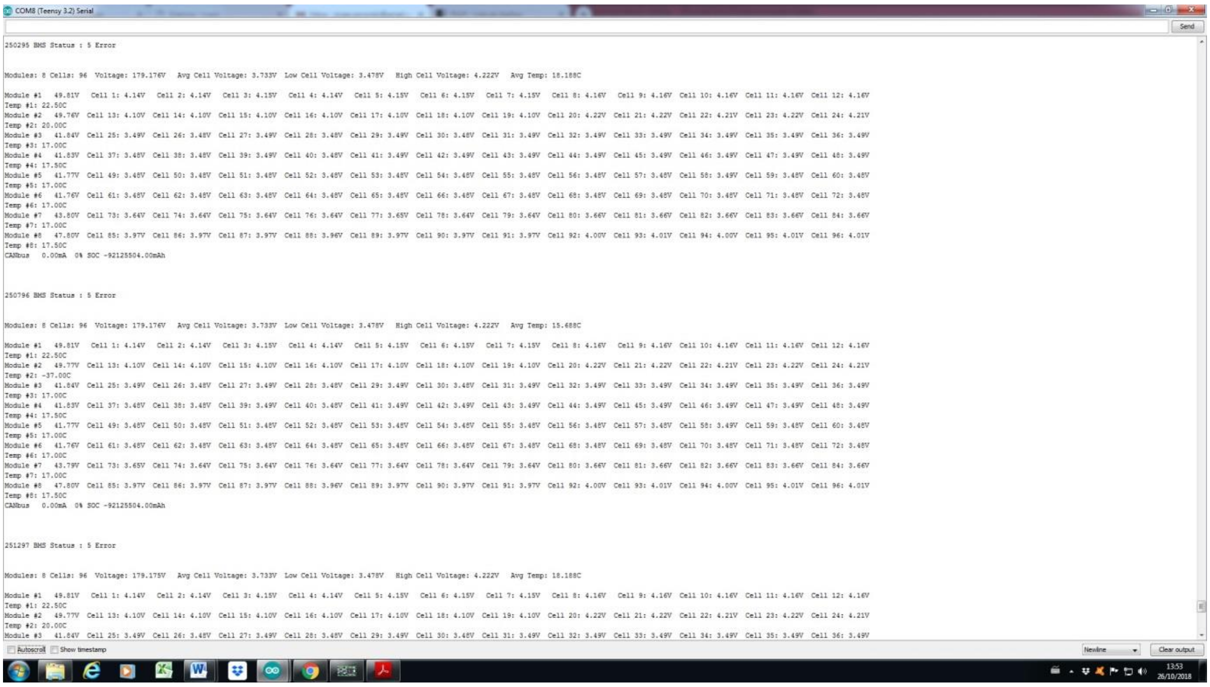
I managed to analyse the module in question and, like it appears in the terminal, had the lowest voltage value in total and the cell imbalance was over 0.25V, as can be seen in the monitor below with the debug settings turned on.



```
2 Temp Found21301,0x18FED502,false,5, 0x0C, 0x00, 0x00, 0x00, 0x00

30 Temp Found21302,0x16A9541E,false,8, 0x80, 0xA8, 0x80, 0xAA, 0x00, 0x00, 0xAA, 0xAA
21302,0x1A555464,false,8, 0x80, 0x34, 0x31, 0x48, 0x20, 0x16, 0x00, 0x00

2 Temp Found21312,0x18FED802,false,5, 0x0C, 0x00, 0x00, 0x00, 0x00

33 Temp Found21314,0x16A95421,false,8, 0x80, 0xA8, 0x80, 0xAA, 0x00, 0x00, 0xAA, 0xAA
21314,0x1A555467,false,8, 0x80, 0x34, 0x31, 0x48, 0x20, 0xAB, 0x00, 0x00



21341 BMS Status : 5 Error


Modules: 8 Cells: 96  Voltage: 170.882V   Avg Cell Voltage: 3.560V  Low Cell Voltage: 3.287V   High Cell Voltage: 4.033V   Avg Temp: 15.688C

Module #1   41.63V   Cell 0: 3.46V   Cell 1: 3.47V   Cell 2: 3.47V   Cell 3: 3.47V   Cell 4: 3.45V   Cell 5: 3.47V   Cell 6: 3.47V   Cell 7: 3.48V   Cell 8: 3.47V   Cell 9: 3.47V  Cell 10: 3.48V  Cell 11: 3.48V
 Temp 1: 17.50C Temp 2: 0.00C Temp 3: 0.00C
Module #2   40.55V  Cell 12: 3.29V  Cell 13: 3.30V  Cell 14: 3.29V  Cell 15: 3.30V  Cell 16: 3.31V  Cell 17: 3.29V  Cell 18: 3.32V  Cell 19: 3.49V  Cell 20: 3.49V  Cell 21: 3.49V  Cell 22: 3.49V  Cell 23: 3.49V
 Temp 1: -37.00C Temp 2: 0.00C Temp 3: 0.00C
Module #3   41.94V  Cell 24: 3.49V  Cell 25: 3.49V  Cell 26: 3.49V  Cell 27: 3.49V  Cell 28: 3.49V  Cell 29: 3.49V  Cell 30: 3.49V  Cell 31: 3.50V  Cell 32: 3.50V  Cell 33: 3.50V  Cell 34: 3.50V  Cell 35: 3.50V
 Temp 1: 17.00C Temp 2: 0.00C Temp 3: 0.00C
Module #4   41.93V  Cell 36: 3.49V  Cell 37: 3.49V  Cell 38: 3.49V  Cell 39: 3.49V  Cell 40: 3.49V  Cell 41: 3.49V  Cell 42: 3.49V  Cell 43: 3.49V  Cell 44: 3.50V  Cell 45: 3.49V  Cell 46: 3.50V  Cell 47: 3.50V
 Temp 1: 18.00C Temp 2: 0.00C Temp 3: 0.00C
Module #5   41.87V  Cell 48: 3.49V  Cell 49: 3.49V  Cell 50: 3.49V  Cell 51: 3.49V  Cell 52: 3.49V  Cell 53: 3.49V  Cell 54: 3.49V  Cell 55: 3.49V  Cell 56: 3.49V  Cell 57: 3.49V  Cell 58: 3.49V  Cell 59: 3.49V
 Temp 1: 18.00C Temp 2: 0.00C Temp 3: 0.00C
Module #6   41.85V  Cell 60: 3.49V  Cell 61: 3.49V  Cell 62: 3.49V  Cell 63: 3.49V  Cell 64: 3.49V  Cell 65: 3.49V  Cell 66: 3.49V  Cell 67: 3.49V  Cell 68: 3.49V  Cell 69: 3.49V  Cell 70: 3.49V  Cell 71: 3.49V
 Temp 1: 18.50C Temp 2: 0.00C Temp 3: 0.00C
Module #7   43.90V  Cell 72: 3.65V  Cell 73: 3.65V  Cell 74: 3.65V  Cell 75: 3.65V  Cell 76: 3.65V  Cell 77: 3.65V  Cell 78: 3.65V  Cell 79: 3.67V  Cell 80: 3.67V  Cell 81: 3.67V  Cell 82: 3.67V  Cell 83: 3.67V
 Temp 1: 18.50C Temp 2: 0.00C Temp 3: 0.00C
Module #8   48.10V  Cell 84: 3.99V  Cell 85: 3.99V  Cell 86: 3.99V  Cell 87: 3.99V  Cell 88: 3.99V  Cell 89: 4.00V  Cell 90: 3.99V  Cell 91: 4.03V  Cell 92: 4.03V  Cell 93: 4.03V  Cell 94: 4.03V  Cell 95: 4.03V
 Temp 1: 18.00C Temp 2: 0.00C Temp 3: 0.00C
CANbus   0.00mA  0% SOC -351094016.00mAh

1,0

176 Temp Found21378,0x1B0,false,8, 0x28, 0xD1, 0x99, 0xA1, 0x00, 0x00, 0xA3, 0x09

7,0

200 Temp Found21378,0x1C8,false,8, 0x32, 0xEF, 0xA5, 0x5C, 0x00, 0x00, 0x5B, 0x0A

4,0

188 Temp Found21379,0x1BC,false,8, 0x7D, 0xEA, 0x9B, 0xBD, 0x00, 0x00, 0xBD, 0x09

3,0

184 Temp Found21379,0x1B8,false,8, 0x32, 0xE7, 0x9B, 0xBD, 0x00, 0x00, 0xBD, 0x09

6,0

196 Temp Found21380,0x1C4,false,8, 0x04, 0x66, 0x9B, 0xB7, 0x00, 0x00, 0xB7, 0x09

2,0

180 Temp Found21381,0x1B4,false,8, 0x03, 0xF2, 0x8E, 0xF9, 0x00, 0x00, 0xF9, 0x08
```

Figure 37 - Debug screen

From Figure 37 and 38 we gather that in the module #2 it is possible to check there is an outlier value of -37.00C in the temperature in alternated measurements. At first, it was hypothesised that this could be due explained by low voltage, most specifically, low cell voltage in the first ones generating an imbalance between the others. After charging the module #2 and waiting for the balancing to take effect I switched the battery to one with the same ID set. This was to eliminate the possibility that the sensor in the original battery, which was already charged, was not damaged. The error persisted and kept giving the outlier value. This kept happening even after switching ports. The conclusion I reached was the issue was not in the battery itself but how the software is interpreting the given value. Furthermore, this overcharging would not have occurred in a practical situation of being used in the electrical vehicle as originally intended since the cells would discharge and charged at the same rate, having no need to wait for days for the balancing to be complete.



```
250295 BMS Status : 5 Error


Modules: 8 Cells: 96  Voltage: 179.176V   Avg Cell Voltage: 3.733V  Low Cell Voltage: 3.478V   High Cell Voltage: 4.222V   Avg Temp: 18.188C

Module #1   49.81V   Cell 1: 4.14V   Cell 2: 4.14V   Cell 3: 4.15V   Cell 4: 4.14V   Cell 5: 4.15V   Cell 6: 4.15V   Cell 7: 4.15V   Cell 8: 4.16V   Cell 9: 4.16V  Cell 10: 4.16V  Cell 11: 4.16V  Cell 12: 4.16V
Temp #1: 22.50C
Module #2   49.76V  Cell 13: 4.10V  Cell 14: 4.10V  Cell 15: 4.10V  Cell 16: 4.10V  Cell 17: 4.10V  Cell 18: 4.10V  Cell 19: 4.10V  Cell 20: 4.22V  Cell 21: 4.22V  Cell 22: 4.21V  Cell 23: 4.22V  Cell 24: 4.21V
Temp #2: 20.00C
Module #3   41.84V  Cell 25: 3.49V  Cell 26: 3.48V  Cell 27: 3.49V  Cell 28: 3.48V  Cell 29: 3.49V  Cell 30: 3.48V  Cell 31: 3.49V  Cell 32: 3.49V  Cell 33: 3.49V  Cell 34: 3.49V  Cell 35: 3.49V  Cell 36: 3.49V
Temp #3: 17.00C
Module #4   41.83V  Cell 37: 3.48V  Cell 38: 3.48V  Cell 39: 3.49V  Cell 40: 3.48V  Cell 41: 3.49V  Cell 42: 3.49V  Cell 43: 3.49V  Cell 44: 3.49V  Cell 45: 3.49V  Cell 46: 3.49V  Cell 47: 3.49V  Cell 48: 3.49V
Temp #4: 17.50C
Module #5   41.77V  Cell 49: 3.48V  Cell 50: 3.48V  Cell 51: 3.48V  Cell 52: 3.48V  Cell 53: 3.48V  Cell 54: 3.48V  Cell 55: 3.48V  Cell 56: 3.48V  Cell 57: 3.48V  Cell 58: 3.49V  Cell 59: 3.48V  Cell 60: 3.48V
Temp #5: 17.00C
Module #6   41.76V  Cell 61: 3.48V  Cell 62: 3.48V  Cell 63: 3.48V  Cell 64: 3.48V  Cell 65: 3.48V  Cell 66: 3.48V  Cell 67: 3.48V  Cell 68: 3.48V  Cell 69: 3.48V  Cell 70: 3.48V  Cell 71: 3.48V  Cell 72: 3.48V
Temp #6: 17.00C
Module #7   43.80V  Cell 73: 3.64V  Cell 74: 3.64V  Cell 75: 3.64V  Cell 76: 3.64V  Cell 77: 3.65V  Cell 78: 3.64V  Cell 79: 3.64V  Cell 80: 3.66V  Cell 81: 3.66V  Cell 82: 3.66V  Cell 83: 3.66V  Cell 84: 3.66V
Temp #7: 17.00C
Module #8   47.80V  Cell 85: 3.97V  Cell 86: 3.97V  Cell 87: 3.97V  Cell 88: 3.96V  Cell 89: 3.97V  Cell 90: 3.97V  Cell 91: 3.97V  Cell 92: 4.00V  Cell 93: 4.01V  Cell 94: 4.00V  Cell 95: 4.01V  Cell 96: 4.01V
Temp #8: 17.50C
CANbus   0.00mA  0% SOC -92125504.00mAh



250796 BMS Status : 5 Error


Modules: 8 Cells: 96  Voltage: 179.176V   Avg Cell Voltage: 3.733V  Low Cell Voltage: 3.478V   High Cell Voltage: 4.222V   Avg Temp: 15.688C

Module #1   49.81V   Cell 1: 4.14V   Cell 2: 4.14V   Cell 3: 4.15V   Cell 4: 4.14V   Cell 5: 4.15V   Cell 6: 4.15V   Cell 7: 4.15V   Cell 8: 4.16V   Cell 9: 4.16V  Cell 10: 4.16V  Cell 11: 4.16V  Cell 12: 4.16V
Temp #1: 22.50C
Module #2   49.77V  Cell 13: 4.10V  Cell 14: 4.10V  Cell 15: 4.10V  Cell 16: 4.10V  Cell 17: 4.10V  Cell 18: 4.10V  Cell 19: 4.10V  Cell 20: 4.22V  Cell 21: 4.22V  Cell 22: 4.21V  Cell 23: 4.22V  Cell 24: 4.21V
Temp #2: -37.00C
Module #3   41.84V  Cell 25: 3.49V  Cell 26: 3.48V  Cell 27: 3.49V  Cell 28: 3.48V  Cell 29: 3.49V  Cell 30: 3.48V  Cell 31: 3.49V  Cell 32: 3.49V  Cell 33: 3.49V  Cell 34: 3.49V  Cell 35: 3.49V  Cell 36: 3.49V
Temp #3: 17.00C
Module #4   41.83V  Cell 37: 3.48V  Cell 38: 3.48V  Cell 39: 3.49V  Cell 40: 3.48V  Cell 41: 3.49V  Cell 42: 3.49V  Cell 43: 3.49V  Cell 44: 3.49V  Cell 45: 3.49V  Cell 46: 3.49V  Cell 47: 3.49V  Cell 48: 3.49V
Temp #4: 17.50C
Module #5   41.77V  Cell 49: 3.48V  Cell 50: 3.48V  Cell 51: 3.48V  Cell 52: 3.48V  Cell 53: 3.48V  Cell 54: 3.48V  Cell 55: 3.48V  Cell 56: 3.48V  Cell 57: 3.48V  Cell 58: 3.49V  Cell 59: 3.48V  Cell 60: 3.48V
Temp #5: 17.00C
Module #6   41.76V  Cell 61: 3.48V  Cell 62: 3.48V  Cell 63: 3.48V  Cell 64: 3.48V  Cell 65: 3.48V  Cell 66: 3.48V  Cell 67: 3.48V  Cell 68: 3.48V  Cell 69: 3.48V  Cell 70: 3.48V  Cell 71: 3.48V  Cell 72: 3.48V
Temp #6: 17.00C
Module #7   43.79V  Cell 73: 3.65V  Cell 74: 3.64V  Cell 75: 3.64V  Cell 76: 3.64V  Cell 77: 3.64V  Cell 78: 3.64V  Cell 79: 3.64V  Cell 80: 3.66V  Cell 81: 3.66V  Cell 82: 3.66V  Cell 83: 3.66V  Cell 84: 3.66V
Temp #7: 17.00C
Module #8   47.80V  Cell 85: 3.97V  Cell 86: 3.97V  Cell 87: 3.97V  Cell 88: 3.96V  Cell 89: 3.97V  Cell 90: 3.97V  Cell 91: 3.97V  Cell 92: 4.00V  Cell 93: 4.01V  Cell 94: 4.00V  Cell 95: 4.01V  Cell 96: 4.01V
Temp #8: 17.50C
CANbus   0.00mA  0% SOC -92125504.00mAh



251297 BMS Status : 5 Error


Modules: 8 Cells: 96  Voltage: 179.175V   Avg Cell Voltage: 3.733V  Low Cell Voltage: 3.478V   High Cell Voltage: 4.222V   Avg Temp: 18.188C

Module #1   49.81V   Cell 1: 4.14V   Cell 2: 4.14V   Cell 3: 4.15V   Cell 4: 4.14V   Cell 5: 4.15V   Cell 6: 4.15V   Cell 7: 4.15V   Cell 8: 4.16V   Cell 9: 4.16V  Cell 10: 4.16V  Cell 11: 4.16V  Cell 12: 4.16V
Temp #1: 22.50C
Module #2   49.77V  Cell 13: 4.10V  Cell 14: 4.10V  Cell 15: 4.10V  Cell 16: 4.10V  Cell 17: 4.10V  Cell 18: 4.10V  Cell 19: 4.10V  Cell 20: 4.22V  Cell 21: 4.22V  Cell 22: 4.21V  Cell 23: 4.22V  Cell 24: 4.21V
Temp #2: 20.00C
Module #3   41.84V  Cell 25: 3.49V  Cell 26: 3.48V  Cell 27: 3.49V  Cell 28: 3.48V  Cell 29: 3.49V  Cell 30: 3.48V  Cell 31: 3.49V  Cell 32: 3.49V  Cell 33: 3.49V  Cell 34: 3.49V  Cell 35: 3.49V  Cell 36: 3.49V
```

Figure 38 - Menu with "Error" status due to Overvoltage

After a week of discharge, there was no significant voltage drop so I decided to connect it to a 40W load as to speed up the process as can be seen in Figure 39.

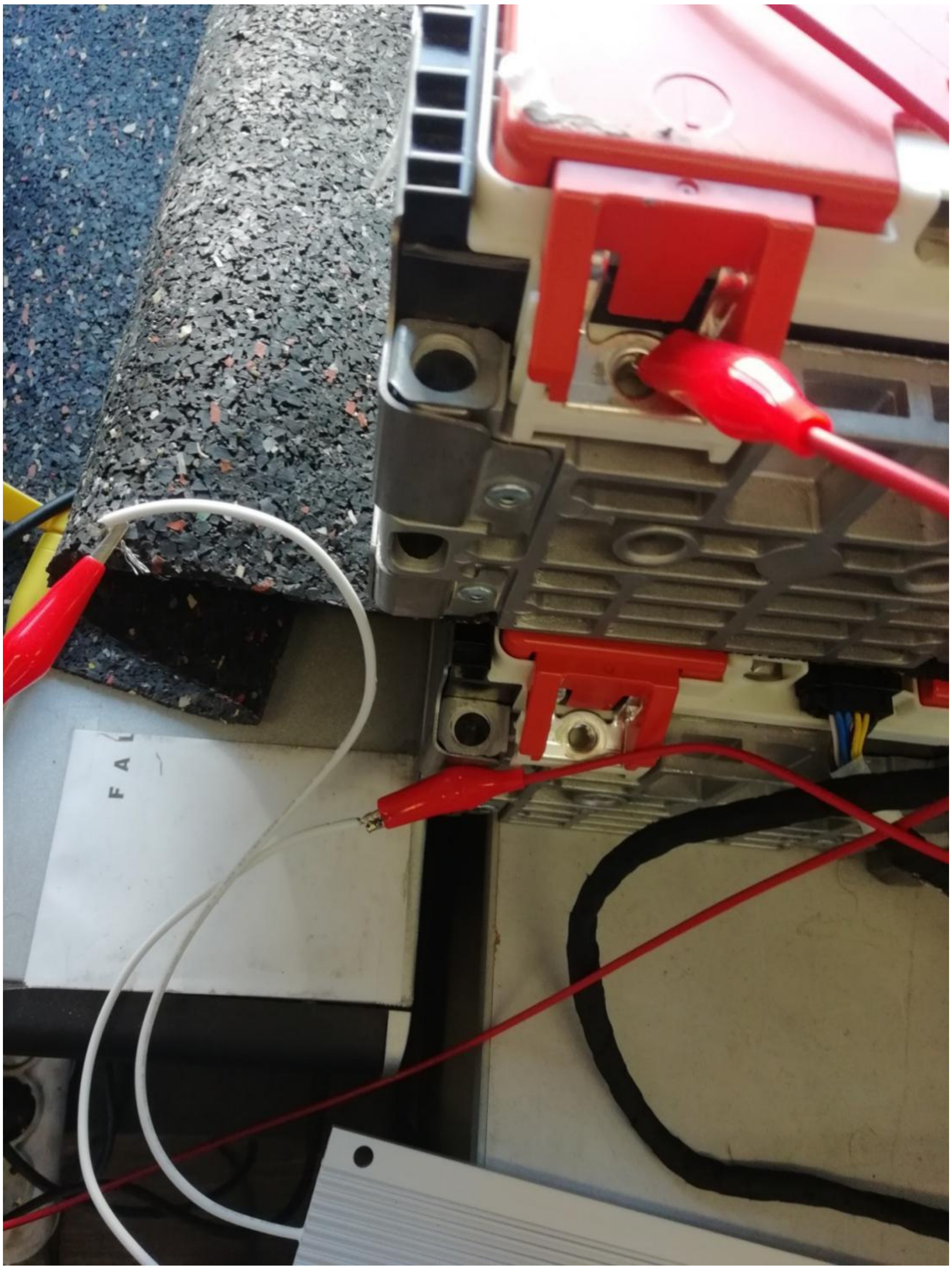


Figure 39 - Discharge process

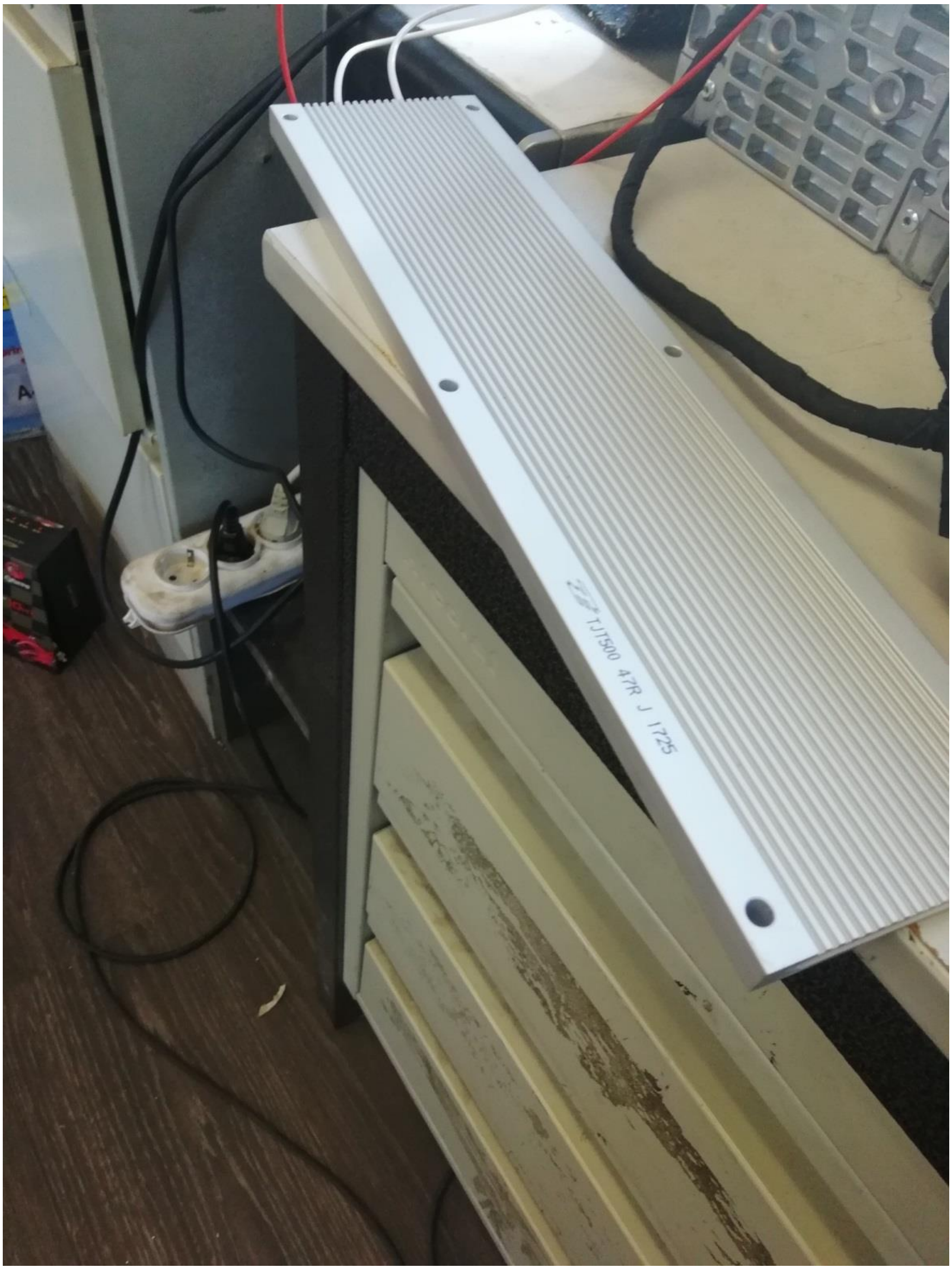
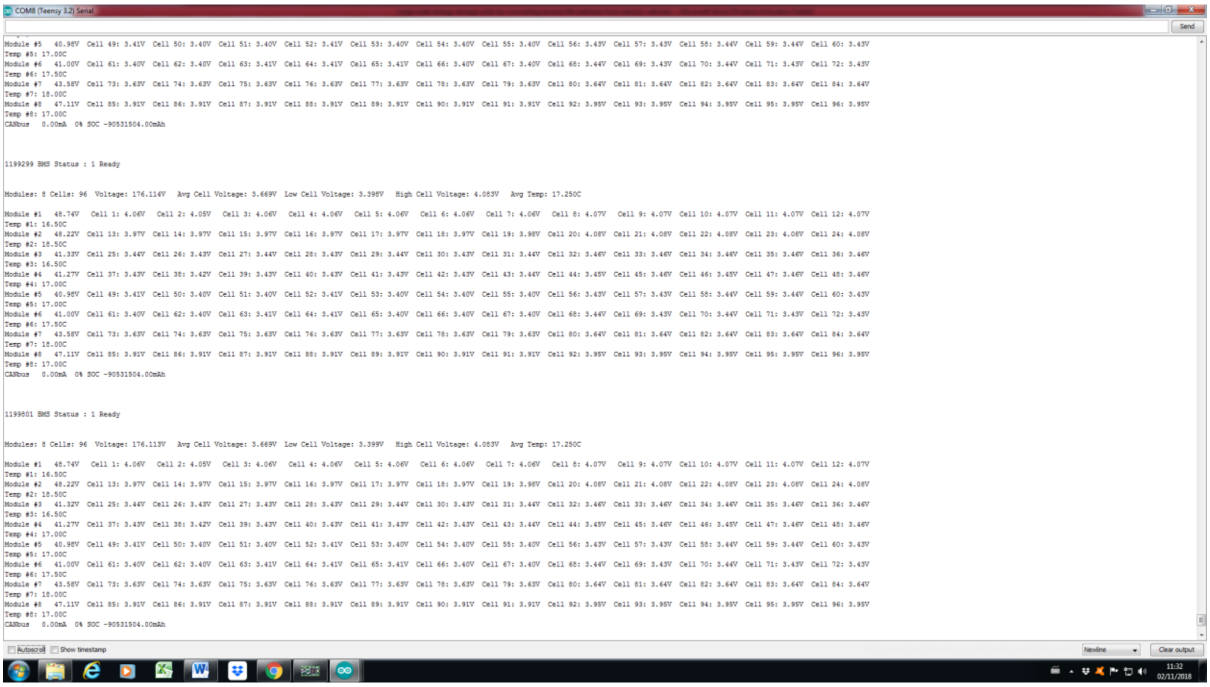


Figure 40 - 40W load

This discharging process, shown in Figure 40, yielded results in the span of an hour and the highest cell dropped below 4.1V meaning there was no more Overvoltage status. It can be inferred that since this was an abnormal situation it would not require that long to discharge by normal means (no load).



```
Module #5   40.98V  Cell 49: 3.41V  Cell 50: 3.40V  Cell 51: 3.40V  Cell 52: 3.41V  Cell 53: 3.40V  Cell 54: 3.40V  Cell 55: 3.40V  Cell 56: 3.43V  Cell 57: 3.43V  Cell 58: 3.44V  Cell 59: 3.44V  Cell 60: 3.43V
Temp #5: 17.00C
Module #6   41.00V  Cell 61: 3.40V  Cell 62: 3.40V  Cell 63: 3.41V  Cell 64: 3.41V  Cell 65: 3.41V  Cell 66: 3.40V  Cell 67: 3.40V  Cell 68: 3.44V  Cell 69: 3.43V  Cell 70: 3.44V  Cell 71: 3.43V  Cell 72: 3.43V
Temp #6: 17.50C
Module #7   43.58V  Cell 73: 3.63V  Cell 74: 3.63V  Cell 75: 3.63V  Cell 76: 3.63V  Cell 77: 3.63V  Cell 78: 3.63V  Cell 79: 3.63V  Cell 80: 3.64V  Cell 81: 3.64V  Cell 82: 3.64V  Cell 83: 3.64V  Cell 84: 3.64V
Temp #7: 18.00C
Module #8   47.11V  Cell 85: 3.91V  Cell 86: 3.91V  Cell 87: 3.91V  Cell 88: 3.91V  Cell 89: 3.91V  Cell 90: 3.91V  Cell 91: 3.91V  Cell 92: 3.95V  Cell 93: 3.95V  Cell 94: 3.95V  Cell 95: 3.95V  Cell 96: 3.95V
Temp #8: 17.00C
CANbus   0.00mA  0% SOC -90531504.00mAh



1199299 BMS Status : 1 Ready


Modules: 8 Cells: 96  Voltage: 176.114V   Avg Cell Voltage: 3.669V  Low Cell Voltage: 3.398V   High Cell Voltage: 4.083V   Avg Temp: 17.250C

Module #1   48.74V   Cell 1: 4.06V   Cell 2: 4.05V   Cell 3: 4.06V   Cell 4: 4.06V   Cell 5: 4.06V   Cell 6: 4.06V   Cell 7: 4.06V   Cell 8: 4.07V   Cell 9: 4.07V  Cell 10: 4.07V  Cell 11: 4.07V  Cell 12: 4.07V
Temp #1: 16.50C
Module #2   48.22V  Cell 13: 3.97V  Cell 14: 3.97V  Cell 15: 3.97V  Cell 16: 3.97V  Cell 17: 3.97V  Cell 18: 3.97V  Cell 19: 3.98V  Cell 20: 4.08V  Cell 21: 4.08V  Cell 22: 4.08V  Cell 23: 4.08V  Cell 24: 4.08V
Temp #2: 18.50C
Module #3   41.33V  Cell 25: 3.44V  Cell 26: 3.43V  Cell 27: 3.44V  Cell 28: 3.43V  Cell 29: 3.44V  Cell 30: 3.43V  Cell 31: 3.44V  Cell 32: 3.46V  Cell 33: 3.46V  Cell 34: 3.46V  Cell 35: 3.46V  Cell 36: 3.46V
Temp #3: 16.50C
Module #4   41.27V  Cell 37: 3.43V  Cell 38: 3.42V  Cell 39: 3.43V  Cell 40: 3.43V  Cell 41: 3.43V  Cell 42: 3.43V  Cell 43: 3.44V  Cell 44: 3.45V  Cell 45: 3.46V  Cell 46: 3.45V  Cell 47: 3.46V  Cell 48: 3.46V
Temp #4: 17.00C
Module #5   40.98V  Cell 49: 3.41V  Cell 50: 3.40V  Cell 51: 3.40V  Cell 52: 3.41V  Cell 53: 3.40V  Cell 54: 3.40V  Cell 55: 3.40V  Cell 56: 3.43V  Cell 57: 3.43V  Cell 58: 3.44V  Cell 59: 3.44V  Cell 60: 3.43V
Temp #5: 17.00C
Module #6   41.00V  Cell 61: 3.40V  Cell 62: 3.40V  Cell 63: 3.41V  Cell 64: 3.41V  Cell 65: 3.40V  Cell 66: 3.40V  Cell 67: 3.40V  Cell 68: 3.44V  Cell 69: 3.43V  Cell 70: 3.44V  Cell 71: 3.43V  Cell 72: 3.43V
Temp #6: 17.50C
Module #7   43.58V  Cell 73: 3.63V  Cell 74: 3.63V  Cell 75: 3.63V  Cell 76: 3.63V  Cell 77: 3.63V  Cell 78: 3.63V  Cell 79: 3.63V  Cell 80: 3.64V  Cell 81: 3.64V  Cell 82: 3.64V  Cell 83: 3.64V  Cell 84: 3.64V
Temp #7: 18.00C
Module #8   47.11V  Cell 85: 3.91V  Cell 86: 3.91V  Cell 87: 3.91V  Cell 88: 3.91V  Cell 89: 3.91V  Cell 90: 3.91V  Cell 91: 3.91V  Cell 92: 3.95V  Cell 93: 3.95V  Cell 94: 3.95V  Cell 95: 3.95V  Cell 96: 3.95V
Temp #8: 17.00C
CANbus   0.00mA  0% SOC -90531504.00mAh



1199801 BMS Status : 1 Ready


Modules: 8 Cells: 96  Voltage: 176.113V   Avg Cell Voltage: 3.669V  Low Cell Voltage: 3.399V   High Cell Voltage: 4.083V   Avg Temp: 17.250C

Module #1   48.74V   Cell 1: 4.06V   Cell 2: 4.05V   Cell 3: 4.06V   Cell 4: 4.06V   Cell 5: 4.06V   Cell 6: 4.06V   Cell 7: 4.06V   Cell 8: 4.07V   Cell 9: 4.07V  Cell 10: 4.07V  Cell 11: 4.07V  Cell 12: 4.07V
Temp #1: 16.50C
Module #2   48.22V  Cell 13: 3.97V  Cell 14: 3.97V  Cell 15: 3.97V  Cell 16: 3.97V  Cell 17: 3.97V  Cell 18: 3.97V  Cell 19: 3.98V  Cell 20: 4.08V  Cell 21: 4.08V  Cell 22: 4.08V  Cell 23: 4.08V  Cell 24: 4.08V
Temp #2: 18.50C
Module #3   41.32V  Cell 25: 3.44V  Cell 26: 3.43V  Cell 27: 3.43V  Cell 28: 3.43V  Cell 29: 3.44V  Cell 30: 3.43V  Cell 31: 3.44V  Cell 32: 3.46V  Cell 33: 3.46V  Cell 34: 3.46V  Cell 35: 3.46V  Cell 36: 3.46V
Temp #3: 16.50C
Module #4   41.27V  Cell 37: 3.43V  Cell 38: 3.42V  Cell 39: 3.43V  Cell 40: 3.43V  Cell 41: 3.43V  Cell 42: 3.43V  Cell 43: 3.44V  Cell 44: 3.45V  Cell 45: 3.46V  Cell 46: 3.45V  Cell 47: 3.46V  Cell 48: 3.46V
Temp #4: 17.00C
Module #5   40.98V  Cell 49: 3.41V  Cell 50: 3.40V  Cell 51: 3.40V  Cell 52: 3.41V  Cell 53: 3.40V  Cell 54: 3.40V  Cell 55: 3.40V  Cell 56: 3.43V  Cell 57: 3.43V  Cell 58: 3.44V  Cell 59: 3.44V  Cell 60: 3.43V
Temp #5: 17.00C
Module #6   41.00V  Cell 61: 3.40V  Cell 62: 3.40V  Cell 63: 3.41V  Cell 64: 3.41V  Cell 65: 3.41V  Cell 66: 3.40V  Cell 67: 3.40V  Cell 68: 3.44V  Cell 69: 3.43V  Cell 70: 3.44V  Cell 71: 3.43V  Cell 72: 3.43V
Temp #6: 17.50C
Module #7   43.58V  Cell 73: 3.63V  Cell 74: 3.63V  Cell 75: 3.63V  Cell 76: 3.63V  Cell 77: 3.63V  Cell 78: 3.63V  Cell 79: 3.63V  Cell 80: 3.64V  Cell 81: 3.64V  Cell 82: 3.64V  Cell 83: 3.64V  Cell 84: 3.64V
Temp #7: 18.00C
Module #8   47.11V  Cell 85: 3.91V  Cell 86: 3.91V  Cell 87: 3.91V  Cell 88: 3.91V  Cell 89: 3.91V  Cell 90: 3.91V  Cell 91: 3.91V  Cell 92: 3.95V  Cell 93: 3.95V  Cell 94: 3.95V  Cell 95: 3.95V  Cell 96: 3.95V
Temp #8: 17.00C
CANbus   0.00mA  0% SOC -90531504.00mAh
```

Figure 41 - Menu with correct values

By analysing the difference in voltages in the cells, there is some difference between the first 7 cells in most modules and the last 5. In figure 41, it is possible to see the BMS structure and there are 2 controllers, each with a maximum of 7 cells being controlled. This means that the first 7 cells have a different controlling unit than the last 5 which might explain the slight offset between the cells. This difference, however, is negligible and causes no detriment to the functioning of the module after this is balanced and the highest cell goes below 4.1V. To sum up, it is possible to see the overall voltage in every individual cell with a total of 96 (12 per module), and the temperature of each module. I modified the code because it was reading 3 temperatures but since there is only 1 sensor in these batteries it was yielding a null value. There's an aggregate of the voltage in the modules, lowest and highest cell voltage and average temperature of the modules. This data is crucial for safety and performance related processes since overvoltage and temperature over the maximum stipulated one can damage the equipment and surroundings heavily and balanced cells are something to thrive for. The testing was now complete.

After some minor modifications on the code and getting temperatures and voltage in each cell, the next step is to optimise the functionality of the batteries, most specifically, their communication protocols as to:

1. Manipulate each cell individually by means of balancing;
2. Test the limits of temperature at which the battery may operate in a storage environment;
3. Control easily and view accurate information in real time.

Chapter 5

Conclusions

This section will wrap up and clear up some possible errors and incomplete features throughout the process and propose fixes to be made in the future. Overall, it will be review of the entire process thus far.

5.1 Practical usage and implementation

As previously mentioned throughout this document, there is a pressing problem to be solved with the integration of DER and RES for two main reasons: impossibility of controlling weather and climate conditions and instability on the grid. This implies that a swift and accurate control equipment in tune with the storage unit that can regulate the strain on the grid of production and consumption without compromising continuous usage is an essential gear in the entire process.

From the perspective of a user it is important to have a product that can be used in a fast and manageable way. As referred previously, this project and, consequently, the product that was delivered at the end of it, has a real-life use case for the new paradigm of RES and DER with more independence and control to the end-user, increasing profits and reducing energy consumption without compromising quality of life and reliability and safety. This interface and electrical installation that is connected to it is, then, a solution for a problem in the world.

The project could not continue at the time because New Electric did not yet have an inverter, that is supposed to be delivered by another company (partnership). However, I managed to create a working product by studying the BMS operation and communication modes and then CAN connections and, by methods of trial and error and reverse engineering, achieved positive results. Since most of the information had to be gathered and analysed from scratch and most of the technical and theoretical practices were complementary to what I learned in the course, I had to learn and try different possibilities and get to a solution.

5.2 Future work

- Improve functionalities of the interface: This can be done by continue programming and adding features to the interface without forgetting the ease of use required for this particular setting. Said improvements range from more information (individual cell values) to better visual.

- Connection to the inverter: The final part of the project will require using the battery pack and display setup as storage unit in an industrial or housing facility. The aforementioned unit requires an inverter for grid connection that was not available at the time I did my project.

As for the batteries that can be used, the program and communication protocols are the same for VW, Tesla and Panasonic, at least. The configuration and schematics may vary in other batteries, but the essence is the same as well as possible connection to the grid meaning the interface can easily be adapted for other modules and be improved on them as well with further control of modules and even individual cells and bidirectionality.

References

[1] *Electrifying the future*. Retrieved September 28, 2018, from Rolls-Royce: https://www.rolls-royce.com/media/our-stories/electrification.aspx#e-fan-x

[2] Bree, T. d. (2018). *VWBMSV2 and TeslaBMSV2.*

[3] Tesla. (2018), *PowerWall*. Retrieved October 29, 2018, from https://www.tesla.com/powerwall/

[4] IEA. *Energy storage Tracking Clean Energy Progress*, Retrieved October 29, 2018, from https://www.iea.org/tcep/energyintegration/energystorage/

[5] Lithium Balance. (2016), *s-BMS User Manual Version 2.01.*

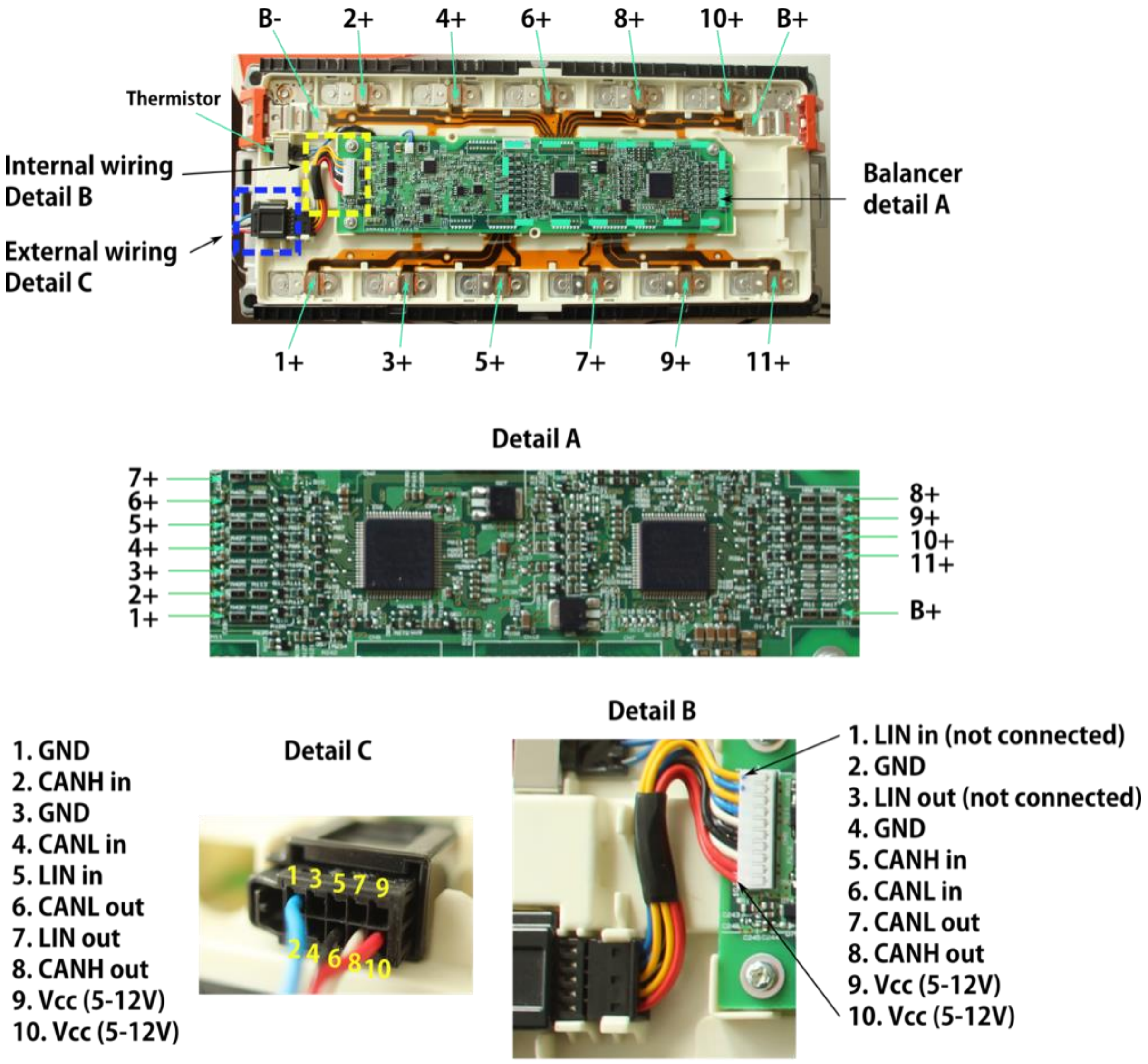
[6] Azure International . (2018). *Energy Storage World Markets Report*.

[7] *Global EV Outlook 2018*. Retrieved September 28, 2018, from https://www.iea.org/gevo2018/

[8] *WIREs Energy Environ 2013, 2: 86–103 doi: 10.1002/wene.34.*

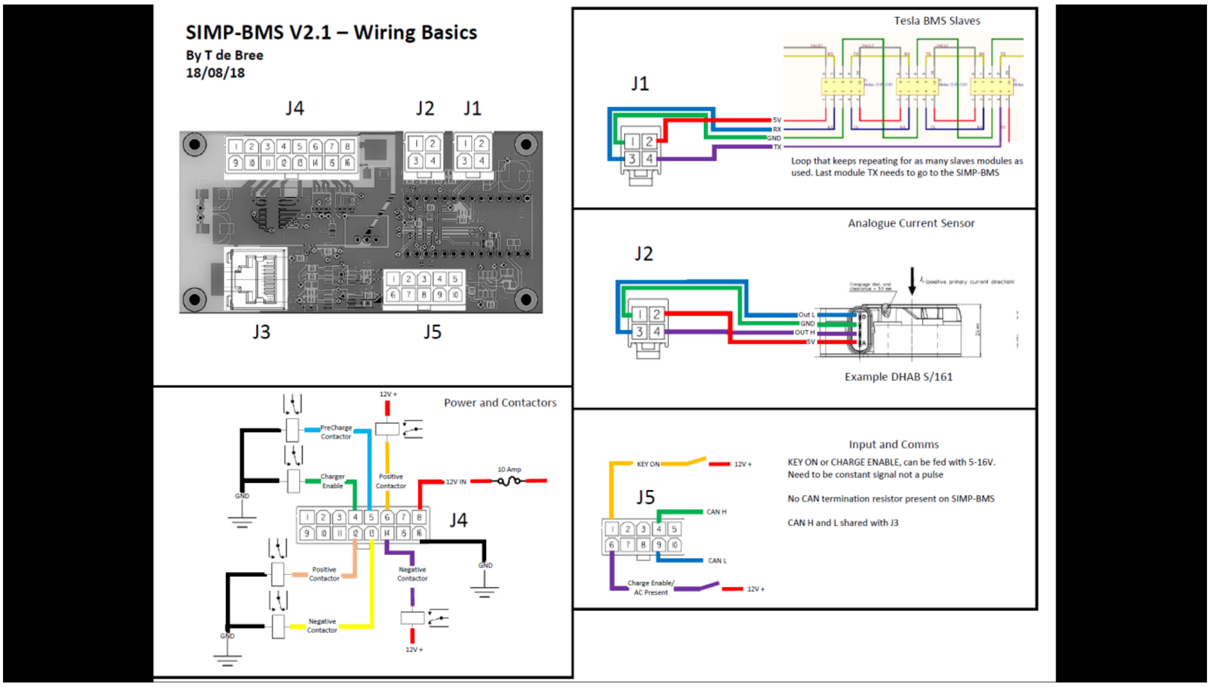
Annexes

A: VW battery, BMS and data connections



CANH in/out, CANL in/out and LIN internally connected

B: Wiring manual for SIMP BMS board



SIMP-BMS V2.1 – Wiring Basics

By T de Bree
18/08/18

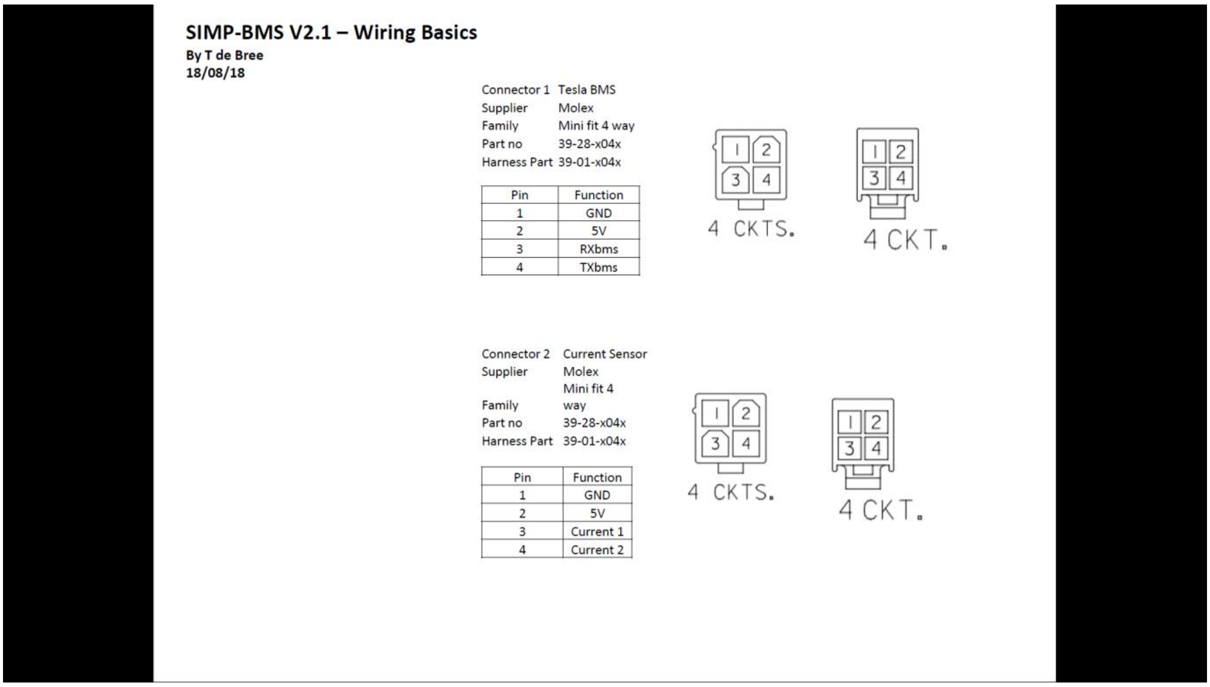
Loop that keeps repeating for as many slaves modules as used. Last module TX needs to go to the SIMP-BMS

Example DHAB S/161

KEY ON or CHARGE ENABLE, can be fed with 5-16V. Need to be constant signal not a pulse

No CAN termination resistor present on SIMP-BMS

CAN H and L shared with J3



SIMP-BMS V2.1 – Wiring Basics

By T de Bree
18/08/18

Connector 1	Tesla BMS
Supplier	Molex
Family	Mini fit 4 way
Part no	39-28-x04x
Harness Part	39-01-x04x

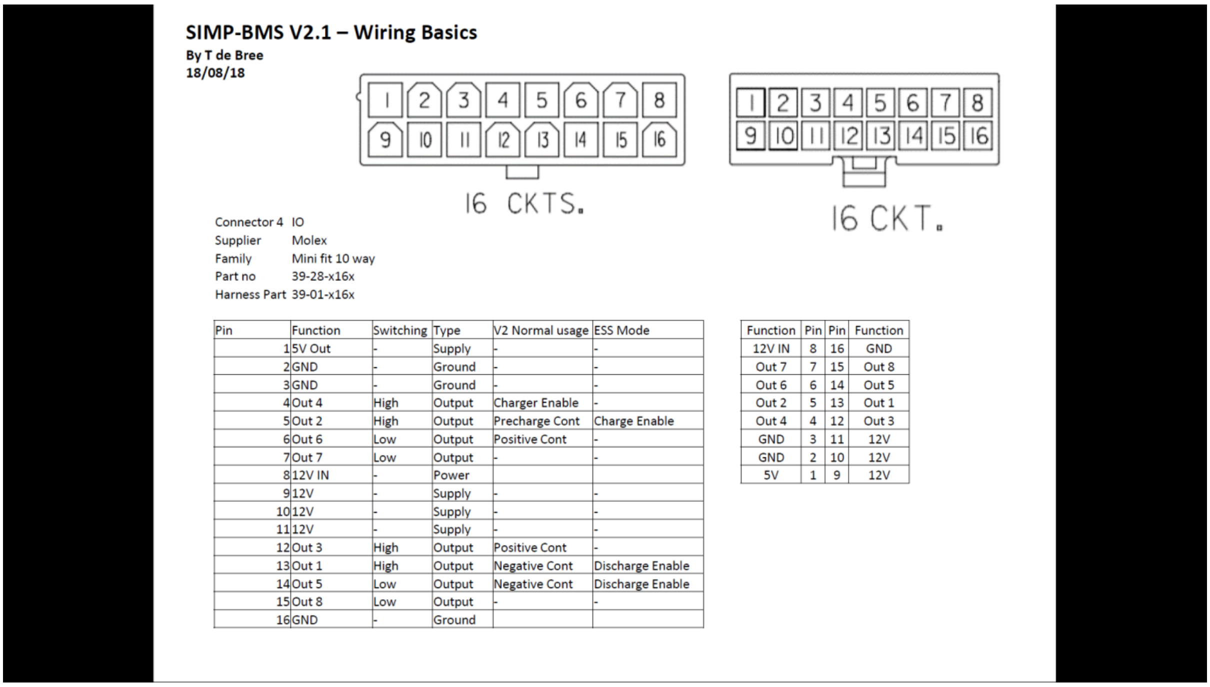
Pin	Function
1	GND
2	5V
3	RXbms
4	TXbms

4 CKTS.
4 CKT.

Connector 2	Current Sensor
Supplier	Molex
Family	Mini fit 4 way
Part no	39-28-x04x
Harness Part	39-01-x04x

Pin	Function
1	GND
2	5V
3	Current 1
4	Current 2

4 CKTS.
4 CKT.



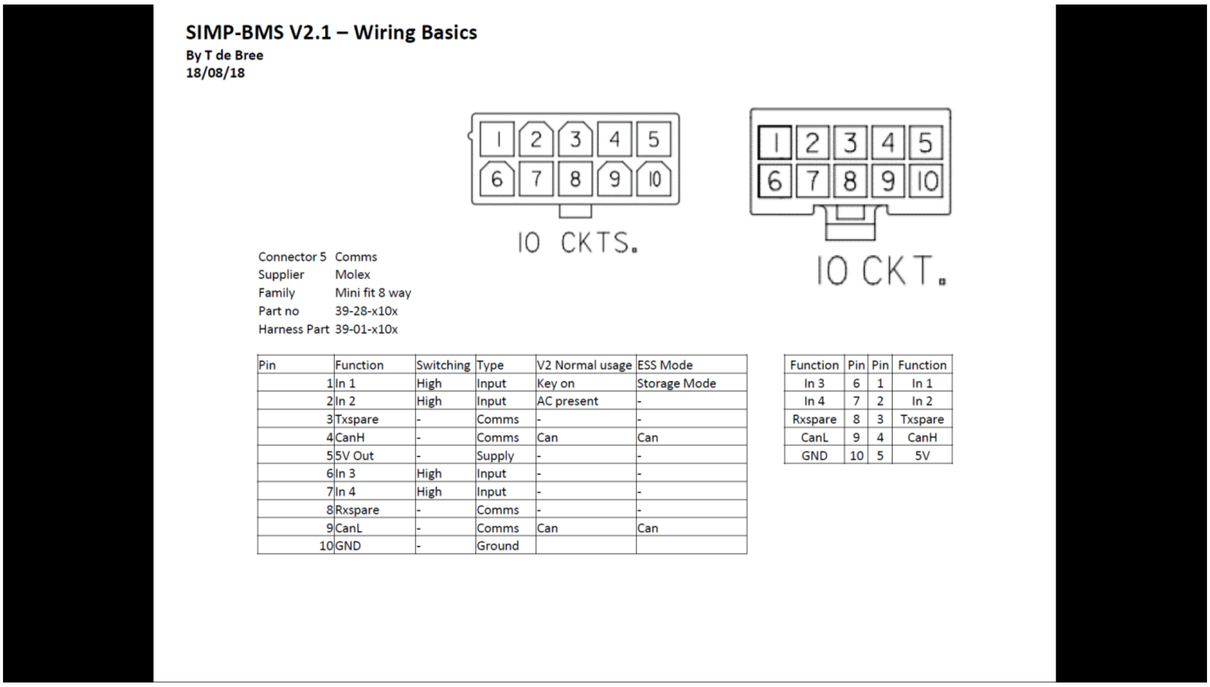
SIMP-BMS V2.1 – Wiring Basics

By T de Bree
18/08/18

Connector 4	IO
Supplier	Molex
Family	Mini fit 10 way
Part no	39-28-x16x
Harness Part	39-01-x16x

Pin	Function	Switching	Type	V2 Normal usage	ESS Mode
1	5V Out	-	Supply	-	-
2	GND	-	Ground	-	-
3	GND	-	Ground	-	-
4	Out 4	High	Output	Charger Enable	-
5	Out 2	High	Output	Precharge Cont	Charge Enable
6	Out 6	Low	Output	Positive Cont	-
7	Out 7	Low	Output	-	-
8	12V IN	-	Power		
9	12V	-	Supply	-	-
10	12V	-	Supply	-	-
11	12V	-	Supply	-	-
12	Out 3	High	Output	Positive Cont	-
13	Out 1	High	Output	Negative Cont	Discharge Enable
14	Out 5	Low	Output	Negative Cont	Discharge Enable
15	Out 8	Low	Output	-	-
16	GND	-	Ground		

Function	Pin	Pin	Function
12V IN	8	16	GND
Out 7	7	15	Out 8
Out 6	6	14	Out 5
Out 2	5	13	Out 1
Out 4	4	12	Out 3
GND	3	11	12V
GND	2	10	12V
5V	1	9	12V



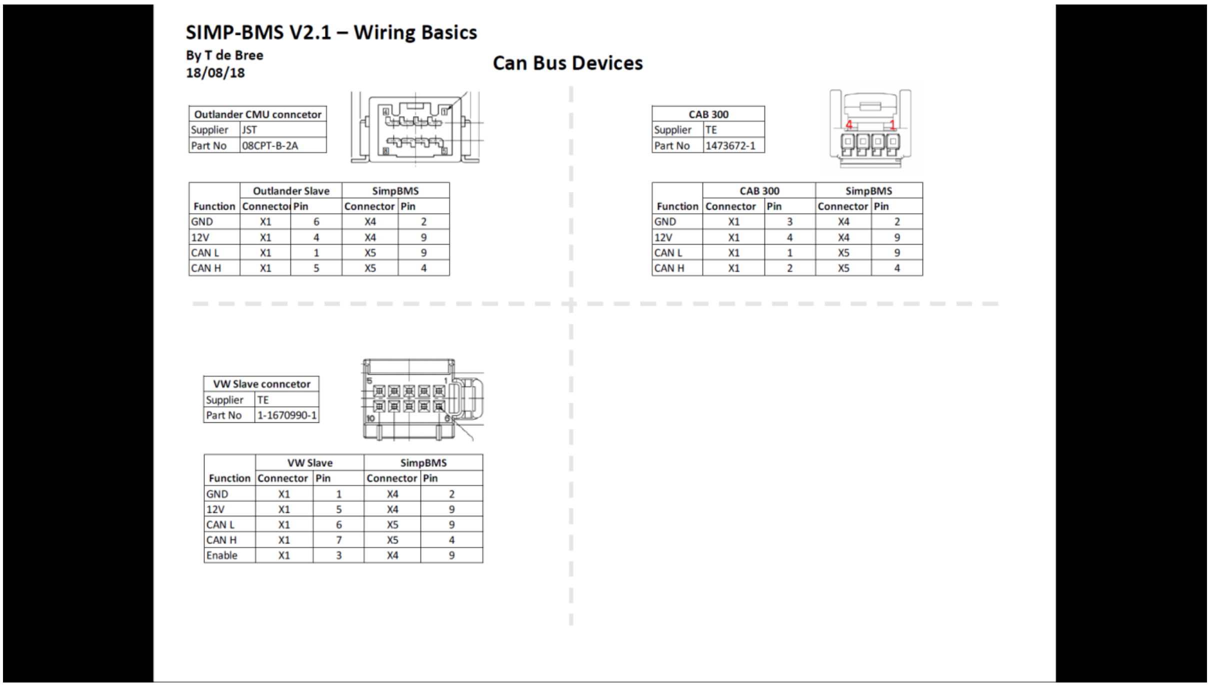
SIMP-BMS V2.1 – Wiring Basics

By T de Bree
18/08/18

Connector 5	Comms
Supplier	Molex
Family	Mini fit 8 way
Part no	39-28-x10x
Harness Part	39-01-x10x

Pin	Function	Switching	Type	V2 Normal usage	ESS Mode
1	In 1	High	Input	Key on	Storage Mode
2	In 2	High	Input	AC present	-
3	Txspare	-	Comms	-	-
4	CanH	-	Comms	Can	Can
5	5V Out	-	Supply	-	-
6	In 3	High	Input	-	-
7	In 4	High	Input	-	-
8	Rxspare	-	Comms	-	-
9	CanL	-	Comms	Can	Can
10	GND	-	Ground		

Function	Pin	Pin	Function
In 3	6	1	In 1
In 4	7	2	In 2
Rxspare	8	3	Txspare
CanL	9	4	CanH
GND	10	5	5V



SIMP-BMS V2.1 – Wiring Basics

By T de Bree
18/08/18

Can Bus Devices

Outlander CMU conncetor	
Supplier	JST
Part No	08CPT-B-2A

	Outlander Slave		SimpBMS	
Function	Connecto	Pin	Connector	Pin
GND	X1	6	X4	2
12V	X1	4	X4	9
CAN L	X1	1	X5	9
CAN H	X1	5	X5	4

CAB 300	
Supplier	TE
Part No	1473672-1

	CAB 300		SimpBMS	
Function	Connector	Pin	Connector	Pin
GND	X1	3	X4	2
12V	X1	4	X4	9
CAN L	X1	1	X5	9
CAN H	X1	2	X5	4

VW Slave conncetor	
Supplier	TE
Part No	1-1670990-1

	VW Slave		SimpBMS	
Function	Connector	Pin	Connector	Pin
GND	X1	1	X4	2
12V	X1	5	X4	9
CAN L	X1	6	X5	9
CAN H	X1	7	X5	4
Enable	X1	3	X4	9

C: VWBMSV2 (January 2019)

```
#include "BMSModuleManager.h"
#include <Arduino.h>
#include "config.h"
#include "SerialConsole.h"
#include "Logger.h"
#include <ADC.h> //https://github.com/pedvide/ADC
#include <EEPROM.h>
#include <FlexCAN.h> //https://github.com/teachop/FlexCAN_Library
#include <SPI.h>
#include <Filters.h>//https://github.com/JonHub/Filters
#include "BMSUtil.h"
#define CPU_REBOOT (_reboot_Teensyduino_());

BMSModuleManager bms;
SerialConsole console;
EEPROMSettings settings;

/////Version Identifier/////////
int firmver = 190113;
//Curent filter//
float filterFrequency = 5.0 ;
FilterOnePole lowpassFilter( LOWPASS, filterFrequency );

//Simple BMS V2 wiring//
const int ACUR1 = A0; // current 1
const int ACUR2 = A1; // current 2
const int IN1 = 17; // input 1 - high active
const int IN2 = 16; // input 2- high active
const int IN3 = 18; // input 1 - high active
const int IN4 = 19; // input 2- high active
const int OUT1 = 11;// output 1 - high active
const int OUT2 = 12;// output 1 - high active
const int OUT3 = 20;// output 1 - high active
const int OUT4 = 21;// output 1 - high active
const int OUT5 = 22;// output 1 - high active
const int OUT6 = 23;// output 1 - high active
const int OUT7 = 5;// output 1 - high active
const int OUT8 = 6;// output 1 - high active
const int led = 13;
const int BMBfault = 11;

byte bmsstatus = 0;
//bms status values
#define Boot 0
#define Ready 1
#define Drive 2
```

```
#define Charge 3
#define Precharge 4
#define Error 5
//
//Current sensor values
#define Undefined 0
#define Analoguedual 1
#define Canbus 2
#define Analoguesing 3
//
//Charger Types
#define NoCharger 0
#define BrusaNLG5 1
#define ChevyVolt 2
#define Eltek 3
#define Elcon 4
//

int Discharge;
//variables for output control
int pulltime = 1000;
int contctrl, contstat = 0; //1 = out 5 high 2 = out 6 high 3 = both
high
unsigned long conttimer1, conttimer2, conttimer3, Pretimer, Pretimer1 =
0;
uint16_t pwmfreq = 15000;//pwm frequency

int pwmcurmax = 200;//Max current to be shown with pwm
int pwmcurmid = 50;//Mid point for pwm dutycycle based on current
int16_t pwmcurmin = 0;//DONOT fill in, calculated later based on other
values

//variables for VE driect bus comms
char* myStrings[] = {"V", "14674", "I", "0", "CE", "-1", "SOC", "800",
"TTG", "-1", "Alarm", "OFF", "Relay", "OFF", "AR", "0", "BMV", "600S",
"FW", "212", "H1", "-3", "H2", "-3", "H3", "0", "H4", "0", "H5", "0",
"H6", "-7", "H7", "13180", "H8", "14774", "H9", "137", "H10", "0",
"H11", "0", "H12", "0"};
//variables for VE can
uint16_t chargevoltage = 49100; //max charge voltage in mv
int chargecurrent;
uint16_t disvoltage = 42000; // max discharge voltage in mv
int discurrent;
uint16_t SOH = 100; // SOH place holder
unsigned char alarm[4], warning[4] = {0, 0, 0, 0};
unsigned char mes[8] = {0, 0, 0, 0, 0, 0, 0, 0};
unsigned char bmsname[8] = {'S', 'I', 'M', 'P', ' ', 'B', 'M', 'S'};
unsigned char bmsmanu[8] = {'T', 'O', 'M', ' ', 'D', 'E', ' ', 'B'};
long unsigned int rxId;
unsigned char len = 0;
byte rxBuf[8];
char msgString[128];                        // Array to store serial
string
uint32_t inbox;
signed long CANmilliamps;
//struct can_frame canMsg;
//MCP2515 CAN1(10); //set CS pin for can controlelr
```

```
//variables for current calulation
int value;
uint16_t offset1 = 1735;
uint16_t offset2 = 1733;
int highconv = 285;
float currentact, RawCur;
float ampsecond;
unsigned long lasttime;
unsigned long looptime, looptime1, UnderTime, cleartime, chargertimer =
0; //ms
int currentsense = 14;
int sensor = 1;
//running average
const int RunningAverageCount = 16;
float RunningAverageBuffer[RunningAverageCount];
int NextRunningAverage;
//Variables for SOC calc
int SOC = 100; //State of Charge
int SOCset = 0;
int SOCtest = 0;
///charger variables
int maxac1 = 16; //Shore power 16A per charger
int maxac2 = 10; //Generator Charging
int chargerid1 = 0x618; //bulk chargers
int chargerid2 = 0x638; //finishing charger
float chargerendbulk = 10.0; //V before Charge Voltage to turn off the
bulk charger/s
float chargerend = 10.0; //V before Charge Voltage to turn off the
finishing charger/s
int chargertoggle = 0;
int ncharger = 1; // number of chargers
//variables
int outputstate = 0;
int incomingByte = 0;
int storagemode = 0;
int x = 0;
int balancecells;
int cellspresent = 0;
int Charged = 0;

//VW BMS CAN variables////////////
int controlid = 0x0BA;
int moduleidstart = 0x1CC;
//Serial Expansion Variables///
int SerialID = 0; //ID assigned over serialbus
int SerialSlaves = 0; //number of slaves present

//Debugging modes//////////////////
int debug = 1;
int gaugedebug = 0;
int inputcheck = 0; //read digital inputs
int outputcheck = 0; //check outputs
int candebug = 0; //view can frames
int debugCur = 0;
int CSVdebug = 0;
int menuload = 0;
int debugdigits = 2; //amount of digits behind decimal for voltage
reading
```

```
ADC *adc = new ADC(); // adc object
void loadSettings()
{
  Logger::console("Resetting to factory defaults");
  settings.version = EEPROM_VERSION;
  settings.checksum = 2;
  settings.canSpeed = 500000;
  settings.batteryID = 0x01; //in the future should be 0xFF to force it
to ask for an address
  settings.OverVSetpoint = 4.1f;
  settings.UnderVSetpoint = 3.0f;
  settings.ChargeVsetpoint = 4.1f;
  settings.ChargeHys = 0.2f; // voltage drop required for charger to
kick back on
  settings.DischVsetpoint = 3.2f;
  settings.CellGap = 0.2f; //max delta between high and low cell
  settings.OverTSetpoint = 65.0f;
  settings.UnderTSetpoint = -10.0f;
  settings.ChargeTSetpoint = 0.0f;
  settings.DisTSetpoint = 40.0f;
  settings.WarnToff = 5.0f; //temp offset before raising warning
  settings.IgnoreTemp = 0; // 0 - use both sensors, 1 or 2 only use
that sensor
  settings.IgnoreVolt = 0.5;//
  settings.balanceVoltage = 3.9f;
  settings.balanceHyst = 0.04f;
  settings.logLevel = 2;
  settings.CAP = 100; //battery size in Ah
  settings.Pstrings = 2; // strings in parallel used to divide voltage
of pack
  settings.Scells = 14;//Cells in series
  settings.discurrentmax = 300; // max discharge current in 0.1A
  settings.chargecurrentmax = 300; //max charge current in 0.1A
  settings.chargecurrentend = 50; //end charge current in 0.1A
  settings.socvolt[0] = 3100; //Voltage and SOC curve for voltage based
SOC calc
  settings.socvolt[1] = 10; //Voltage and SOC curve for voltage based
SOC calc
  settings.socvolt[2] = 4100; //Voltage and SOC curve for voltage based
SOC calc
  settings.socvolt[3] = 90; //Voltage and SOC curve for voltage based
SOC calc
  settings.invertcur = 0; //Invert current sensor direction
  settings.cursens = 2;
  settings.voltsoc = 0; //SOC purely voltage based
  settings.Pretime = 5000; //ms of precharge time
  settings.conthold = 50; //holding duty cycle for contactor 0-255
  settings.Precurrent = 1000; //ma before closing main contator
  settings.Serialexp = 0; //0 standalone - 1 Serial Master - 2 Serial
Slave
  settings.gaugelow = 50; //empty fuel gauge pwm
  settings.gaugehigh = 255; //full fuel gauge pwm
  settings.ESSmode = 0; //activate ESS mode
  settings.chargertype = 2; // 1 - Brusa NLG5xx 2 - Volt charger 0 -No
Charger
  settings.chargerspd = 100; //ms per message
  settings.UnderDur = 5000; //ms of allowed undervoltage before
throwing open stopping discharge.
  settings.CurDead = 5;// mV of dead band on current sensor
}
CAN_message_t msg;
CAN_message_t inMsg;
CAN_filter_t filter;
uint32_t lastUpdate;

void setup()
{
  delay(4000);  //just for easy debugging. It takes a few seconds for
USB to come up properly on most OS's
  pinMode(ACUR1, INPUT);
  pinMode(ACUR2, INPUT);
  pinMode(IN1, INPUT);
  pinMode(IN2, INPUT);
  pinMode(IN3, INPUT);
  pinMode(IN4, INPUT);
  pinMode(OUT1, OUTPUT); // drive contactor
  pinMode(OUT2, OUTPUT); // precharge
  pinMode(OUT3, OUTPUT); // charge relay
  pinMode(OUT4, OUTPUT); // Negative contactor
  pinMode(OUT5, OUTPUT); // pwm driver output
  pinMode(OUT6, OUTPUT); // pwm driver output
  pinMode(OUT7, OUTPUT); // pwm driver output
  pinMode(OUT8, OUTPUT); // pwm driver output
  pinMode(led, OUTPUT);
  analogWriteFrequency(OUT5, pwmfreq);
  analogWriteFrequency(OUT6, pwmfreq);
  analogWriteFrequency(OUT7, pwmfreq);
  analogWriteFrequency(OUT8, pwmfreq);
  Can0.begin(500000);

  //set filters for standard
  for (int i = 0; i < 8; i++)
  {
    Can0.getFilter(filter, i);
    filter.flags.extended = 0;
    Can0.setFilter(filter, i);
  }
  //set filters for extended
  for (int i = 9; i < 13; i++)
  {
    Can0.getFilter(filter, i);
    filter.flags.extended = 1;
    Can0.setFilter(filter, i);
  }
  //if using enable pins on a transceiver they need to be set on
  adc->setAveraging(16); // set number of averages
  adc->setResolution(16); // set bits of resolution
  adc->setConversionSpeed(ADC_CONVERSION_SPEED::MED_SPEED);
  adc->setSamplingSpeed(ADC_SAMPLING_SPEED::MED_SPEED);
  adc->startContinuous(ACUR1, ADC_0);

  SERIALCONSOLE.begin(115200);
  SERIALCONSOLE.println("Starting up!");
  SERIALCONSOLE.println("SimpBMS V2 VW");
  // Display reason the Teensy was last reset
  Serial.println();
  Serial.println("Reason for last Reset: ");
  if (RCM_SRS1 & RCM_SRS1_SACKERR)   Serial.println("Stop Mode
Acknowledge Error Reset");
  if (RCM_SRS1 & RCM_SRS1_MDM_AP)    Serial.println("MDM-AP Reset");
  if (RCM_SRS1 & RCM_SRS1_SW)        Serial.println("Software Reset");
// reboot with SCB_AIRCR = 0x05FA0004
  if (RCM_SRS1 & RCM_SRS1_LOCKUP)    Serial.println("Core Lockup Event
Reset");
  if (RCM_SRS0 & RCM_SRS0_POR)       Serial.println("Power-on Reset");
// removed / applied power
  if (RCM_SRS0 & RCM_SRS0_PIN)       Serial.println("External Pin
Reset");               // Reboot with software download
  if (RCM_SRS0 & RCM_SRS0_WDOG)      Serial.println("Watchdog(COP)
Reset");              // WDT timed out
  if (RCM_SRS0 & RCM_SRS0_LOC)       Serial.println("Loss of External
Clock Reset");
  if (RCM_SRS0 & RCM_SRS0_LOL)       Serial.println("Loss of Lock in
PLL Reset");
  if (RCM_SRS0 & RCM_SRS0_LVD)       Serial.println("Low-voltage Detect
Reset");
  Serial.println();
  ///////////////////
  // enable WDT
  noInterrupts();                                         // don't
allow interrupts while setting up WDOG
  WDOG_UNLOCK = WDOG_UNLOCK_SEQ1;                         // unlock
access to WDOG registers
  WDOG_UNLOCK = WDOG_UNLOCK_SEQ2;
  delayMicroseconds(1);                                   // Need to
wait a bit..
  WDOG_TOVALH = 0x1000;
  WDOG_TOVALL = 0x0000;
  WDOG_PRESC  = 0;
  WDOG_STCTRLH |= WDOG_STCTRLH_ALLOWUPDATE |
                  WDOG_STCTRLH_WDOGEN | WDOG_STCTRLH_WAITEN |
                  WDOG_STCTRLH_STOPEN | WDOG_STCTRLH_CLKSRC;
  interrupts();
  /////////////////
  SERIALBMS.begin(115200);
  //SERIALBMS.begin(612500); //Tesla serial bus
  //VE.begin(19200); //Victron VE direct bus
#if defined (__arm__) && defined (__SAM3X8E__)
  serialSpecialInit(USART0, 612500); //required for Due based boards as
the stock core files don't support 612500 baud.
#endif
  SERIALCONSOLE.println("Started serial interface to BMS.");
  EEPROM.get(0, settings);
  if (settings.version != EEPROM_VERSION)
  {
    loadSettings();
  }
  bms.renumberBoardIDs();
  Logger::setLoglevel(Logger::Off); //Debug = 0, Info = 1, Warn = 2,
Error = 3, Off = 4
  lastUpdate = 0;
  //bms.clearFaults();
  bms.findBoards();
  digitalWrite(led, HIGH);
  bms.setPstrings(settings.Pstrings);
  bms.setSensors(settings.IgnoreTemp, settings.IgnoreVolt);
  ////Calculate fixed numbers
  pwmcurmin = (pwmcurmid / 50 * pwmcurmax * -1);
  ////
  if (settings.Serialexp == 1)
  {
    delay(300);//wait for all other boards to boot
    Serialslaveinit();
  }

  ///precharge timer kickers
  Pretimer = millis();
  Pretimer1  = millis();
}

void loop()
{
  while (Can0.available())
  {
    canread();
  }
  if (SERIALCONSOLE.available() > 0)
  {
    menu();
  }
  if (settings.Serialexp != 0)
  {
    if (SERIALBMS.available() > 0)
    {
      Serialexp();
    }
  }

  if (outputcheck != 1)
  {
    contcon();
    if (settings.ESSmode == 1)
    {
      bmsstatus = Boot;
      contctrl = contctrl | 4; //turn on negative contactor

      if (digitalRead(IN1) == LOW)//Key OFF
      {
        if (storagemode == 1)
        {
          storagemode = 0;
        }
      }
      else
      {
        if (storagemode == 0)
        {
          storagemode = 1;
        }
      }
      if (bms.getHighCellVolt() > settings.balanceVoltage &&
bms.getHighCellVolt() > bms.getLowCellVolt() + settings.balanceHyst)
      {
        balancecells = 1;
      }
      else
      {
        balancecells = 0;
      }
      //Pretimer + settings.Pretime > millis();
      if (storagemode == 1)
      {
        if (bms.getHighCellVolt() > settings.StoreVsetpoint)
        {
          digitalWrite(OUT3, LOW);//turn off charger
          contctrl = contctrl & 253;
          Pretimer = millis();
          Charged = 1;
        }
        else
        {
          if (Charged == 1 && bms.getHighCellVolt() <
(settings.StoreVsetpoint - settings.ChargeHys))
          {
            Charged = 0;
            digitalWrite(OUT3, HIGH);//turn on charger
            if (Pretimer + settings.Pretime < millis())
            {
              contctrl = contctrl | 2;
              Pretimer = 0;
            }
          }
        }
      }
      else
      {
        if (bms.getHighCellVolt() > settings.OverVSetpoint ||
bms.getHighCellVolt() > settings.ChargeVsetpoint)
        {
          digitalWrite(OUT3, LOW);//turn off charger
          contctrl = contctrl & 253;
          Pretimer = millis();
          Charged = 1;
        }
        else
        {
          if (Charged == 1 && bms.getHighCellVolt() <
(settings.ChargeVsetpoint - settings.ChargeHys))
          {
            Charged = 0;
            digitalWrite(OUT3, HIGH);//turn on charger
            if (Pretimer + settings.Pretime < millis())
            {
              // Serial.println();
              //Serial.print(Pretimer);
              contctrl = contctrl | 2;
            }
          }
```

```
    }
   }
   if (bms.getLowCellVolt() < settings.UnderVSetpoint ||
bms.getLowCellVolt() < settings.DischVsetpoint)
   {
    digitalWrite(OUT1, LOW);//turn off discharge
    contctrl = contctrl & 254;
    Pretimer1 = millis();
   }
   else
   {
    digitalWrite(OUT1, HIGH);//turn on discharge
    if (Pretimer1 + settings.Pretime < millis())
    {
     contctrl = contctrl | 1;
    }
   }
   //pwmcomms();
  }
  else
  {
   switch (bmsstatus)
   {
    case (Boot):
     Discharge = 0;
     digitalWrite(OUT3, LOW);//turn off charger
     digitalWrite(OUT1, LOW);//turn off discharge
     contctrl = 0;
     bmsstatus = Ready;
     break;
    case (Ready):
     Discharge = 0;
     if (bms.getHighCellVolt() > settings.balanceVoltage &&
bms.getHighCellVolt() > bms.getLowCellVolt() + settings.balanceHyst)
     {
      bms.balanceCells();
      balancecells = 1;
     }
     else
     {
      balancecells = 0;
     }
     if (digitalRead(IN3) == HIGH && (bms.getHighCellVolt() <
(settings.ChargeVsetpoint - settings.ChargeHys))) //detect AC present
for charging and check not balancing
     {
      bmsstatus = Charge;
     }
     if (digitalRead(IN1) == HIGH) //detect Key ON
     {
      bmsstatus = Precharge;
      Pretimer = millis();
     }
     break;
    case (Precharge):
     Discharge = 0;
     Prechargecon();
     break;
```

```
        case (Drive):
          Discharge = 1;
          if (digitalRead(IN1) == LOW)//Key OFF
          {
            digitalWrite(OUT4, LOW);
            digitalWrite(OUT1, LOW);

            contctrl = 0; //turn off out 5 and 6
            bmsstatus = Ready;
          }
          break;
        case (Charge):
          Discharge = 0;
          digitalWrite(OUT3, HIGH);//enable charger
          if (bms.getHighCellVolt() > settings.balanceVoltage)
          {
            bms.balanceCells();
            balancecells = 1;
          }
          else
          {
            balancecells = 0;
          }
          if (bms.getHighCellVolt() > settings.ChargeVsetpoint)
          {
            digitalWrite(OUT3, LOW);//turn off charger
            bmsstatus = Ready;
          }
          if (digitalRead(IN3) == LOW)//detect AC not present for
charging
          {
            digitalWrite(OUT3, LOW);//turn off charger
            bmsstatus = Ready;
          }
          break;
        case (Error):
          Discharge = 0;
          if (digitalRead(IN3) == HIGH) //detect AC present for
charging
          {
            bmsstatus = Charge;
          }
          if (cellspresent == bms.seriescells()) //detect a fault in
cells detected
          {
            if (bms.getLowCellVolt() >= settings.UnderVSetpoint)
            {
              bmsstatus = Ready;
            }
          }
          break;
      }
    }
    if (settings.cursens == Analoguedual || settings.cursens ==
Analoguesing)
    {
      getcurrent();
    }
```

```
  }
  if (millis() - looptime > 500)
  {
    looptime = millis();
    bms.getAllVoltTemp();
    //UV  check
    if (settings.ESSmode == 1)
    {
      if (bms.getLowCellVolt() < settings.UnderVSetpoint ||
bms.getHighCellVolt() < settings.UnderVSetpoint)
      {
        bmsstatus = Error;
      }
    }
    else //In 'vehicle' mode
    {
      if (bms.getLowCellVolt() < settings.UnderVSetpoint ||
bms.getHighCellVolt() < settings.UnderVSetpoint)
      {
        if (UnderTime > millis()) //check is last time not undervoltage
is longer thatn UnderDur ago
        {
          bmsstatus = Error;
        }
      }
      else
      {
        UnderTime = millis() + settings.UnderDur;
      }
    }

    if (debug != 0)
    {
      printbmsstat();
      bms.printPackDetails(debugdigits);
    }
    if (CSVdebug != 0)
    {
      bms.printAllCSV();
    }
    if (inputcheck != 0)
    {
      inputdebug();
    }
    if (outputcheck != 0)
    {
      outputdebug();
    }
    else
    {
      gaugeupdate();
    }
    updateSOC();
    currentlimit();
    VEcan();
    sendcommand();
    if (cellspresent == 0)
    {
      cellspresent = bms.seriescells();//set amount of connected cells,
might need delay
    }
    else
    {
      if (cellspresent != bms.seriescells()) //detect a fault in cells
detected
      {
        bmsstatus = Error;
      }
    }
    if (settings.Serialexp != 0)
    {
      if (settings.Serialexp == 1)
      {
        SerialReqData();
      }
    }
    alarmupdate();
    resetwdog();
  }
  if (millis() - cleartime > 5000)
  {
    //bms.clearmodules();
  }
  if (millis() - looptime1 > settings.chargerspd)
  {
    looptime1 = millis();
    if (settings.ESSmode == 1)
    {
      chargercomms();
    }
    else
    {
      if (bmsstatus == Charge)
      {
        chargercomms();
      }
    }
  }
}

void alarmupdate()
{
  alarm[0] = 0x00;
  if (settings.OverVSetpoint < bms.getHighCellVolt())
  {
    alarm[0] = 0x04;
  }
  if (bms.getLowCellVolt() < settings.UnderVSetpoint)
  {
    alarm[0] |= 0x10;
  }
  if (bms.getAvgTemperature() > settings.OverTSetpoint)
  {
    alarm[0] |= 0x40;
  }
  alarm[1] = 0;
  if (bms.getAvgTemperature() < settings.UnderTSetpoint)
  {
    alarm[1] = 0x01;
  }
  alarm[3] = 0;
  if ((bms.getHighCellVolt() - bms.getLowCellVolt()) >
settings.CellGap)
  {
    alarm[3] = 0x01;
  }

  ///warnings///
  warning[0] = 0;
  if (bms.getHighCellVolt() > (settings.OverVSetpoint -
settings.WarnOff))
  {
    warning[0] = 0x04;
  }
  if (bms.getLowCellVolt() < (settings.UnderVSetpoint +
settings.WarnOff))
  {
    warning[0] |= 0x10;
  }

  if (bms.getAvgTemperature() > (settings.OverTSetpoint -
settings.WarnToff))
  {
    warning[0] |= 0x40;
  }
  warning[1] = 0;
  if (bms.getAvgTemperature() < (settings.UnderTSetpoint +
settings.WarnToff))
  {
    warning[1] = 0x01;
  }
}

void gaugeupdate()
{
  if (gaugedebug == 1)
  {
    SOCtest = SOCtest + 10;
    if (SOCtest > 1000)
    {
      SOCtest = 0;
    }
    analogWrite(OUT8, map(SOCtest * 0.1, 0, 100, settings.gaugelow,
settings.gaugehigh));
    SERIALCONSOLE.println("  ");
    SERIALCONSOLE.print("SOC : ");
    SERIALCONSOLE.print(SOCtest * 0.1);
    SERIALCONSOLE.print("  fuel pwm : ");
    SERIALCONSOLE.print(map(SOCtest * 0.1, 0, 100, settings.gaugelow,
settings.gaugehigh));
    SERIALCONSOLE.println("  ");
  }
  if (gaugedebug == 2)
  {
    SOCtest = 0;
    analogWrite(OUT8, map(SOCtest * 0.1, 0, 100, settings.gaugelow,
settings.gaugehigh));
  }
  if (gaugedebug == 3)
  {
    SOCtest = 1000;
    analogWrite(OUT8, map(SOCtest * 0.1, 0, 100, settings.gaugelow,
settings.gaugehigh));
  }
  if (gaugedebug == 0)
  {
    analogWrite(OUT8, map(SOC, 0, 100, settings.gaugelow,
settings.gaugehigh));
  }
}
void printbmsstat()
{
  SERIALCONSOLE.println();
  SERIALCONSOLE.println();
  SERIALCONSOLE.println();
  SERIALCONSOLE.print("BMS Status : ");
  if (settings.ESSmode == 1)
  {
    SERIALCONSOLE.print("ESS Mode ");
    if (bms.getLowCellVolt() < settings.UnderVSetpoint)
    {
      SERIALCONSOLE.print(": UnderVoltage ");
    }
    if (bms.getHighCellVolt() > settings.OverVSetpoint)
    {
      SERIALCONSOLE.print(": OverVoltage ");
    }
    if ((bms.getHighCellVolt() - bms.getLowCellVolt()) >
settings.CellGap)
    {
      SERIALCONSOLE.print(": Cell Imbalance ");
    }
    if (bms.getAvgTemperature() > settings.OverTSetpoint)
    {
      SERIALCONSOLE.print(": Over Temp ");
    }
    if (bms.getAvgTemperature() < settings.UnderTSetpoint)
    {
      SERIALCONSOLE.print(": Under Temp ");
    }
    if (storagemode == 1)
    {
      if (bms.getLowCellVolt() > settings.StoreVsetpoint)
      {
        SERIALCONSOLE.print(": OverVoltage Storage ");
        SERIALCONSOLE.print(": UNhappy:");
      }
      else
      {
        SERIALCONSOLE.print(": Happy ");
      }
    }
    else
    {
      if (bms.getLowCellVolt() > settings.UnderVSetpoint &&
bms.getHighCellVolt() < settings.OverVSetpoint)
      {
        if ( bmsstatus == Error)
        {
          SERIALCONSOLE.print(": UNhappy:");
        }
        else
        {
          SERIALCONSOLE.print(": Happy ");
        }
      }
    }
  }
  else
  {
    SERIALCONSOLE.print(bmsstatus);
    switch (bmsstatus)
    {
      case (Boot):
        SERIALCONSOLE.print(" Boot ");
        break;
      case (Ready):
        SERIALCONSOLE.print(" Ready ");
        break;
      case (Precharge):
        SERIALCONSOLE.print(" Precharge ");
        break;
      case (Drive):
        SERIALCONSOLE.print(" Drive ");
        break;
      case (Charge):
        SERIALCONSOLE.print(" Charge ");
        break;
      case (Error):
        SERIALCONSOLE.print(" Error ");
        break;
    }
  }
  SERIALCONSOLE.print("  ");
  if (digitalRead(IN3) == HIGH)
  {
    SERIALCONSOLE.print("| AC Present |");
  }
  if (digitalRead(IN1) == HIGH)
  {
    SERIALCONSOLE.print("| Key ON |");
  }
  if (balancecells == 1)
  {
    SERIALCONSOLE.print("|Balancing Active");
  }
  SERIALCONSOLE.print("  ");
  SERIALCONSOLE.print(cellspresent);
  SERIALCONSOLE.println();
  SERIALCONSOLE.print("Out:");
  SERIALCONSOLE.print(digitalRead(OUT1));
  SERIALCONSOLE.print(digitalRead(OUT2));
  SERIALCONSOLE.print(digitalRead(OUT3));
  SERIALCONSOLE.print(digitalRead(OUT4));
  SERIALCONSOLE.print(" Cont:");
  SERIALCONSOLE.print(contstat, BIN);
  SERIALCONSOLE.print(" In:");
  SERIALCONSOLE.print(digitalRead(IN1));
  SERIALCONSOLE.print(digitalRead(IN2));
  SERIALCONSOLE.print(digitalRead(IN3));
  SERIALCONSOLE.print(digitalRead(IN4));
}

void getcurrent()
{
  if ( settings.cursens == Analoguedual || settings.cursens ==
Analoguesing)
  {
    if ( settings.cursens == Analoguedual)
    {
      if (currentact < 19000 && currentact > -19000)
      {
        sensor = 1;
        adc->startContinuous(ACUR1, ADC_0);
      }
      else
      {
        sensor = 2;
        adc->startContinuous(ACUR2, ADC_0);
      }
    }
    else
    {
      sensor = 1;
      adc->startContinuous(ACUR1, ADC_0);
    }
    if (sensor == 1)
    {
      if (debugCur != 0)
      {
        SERIALCONSOLE.println();
        if ( settings.cursens == Analoguedual)
        {
          SERIALCONSOLE.print("Low Range: ");
        }
        else
        {
          SERIALCONSOLE.print("Single In: ");
        }
        SERIALCONSOLE.print("Value ADC0: ");
      }
      value = (uint16_t)adc->analogReadContinuous(ADC_0); // the
unsigned is necessary for 16 bits, otherwise values larger than 3.3/2 V
are negative!
      if (debugCur != 0)
      {
        SERIALCONSOLE.print(value * 3300 / adc->getMaxValue(ADC_0));
//- settings.offset1)
```

```
        SERIALCONSOLE.print(" ");
        SERIALCONSOLE.print(settings.offset1);
      }
      RawCur = int16_t((value * 3300 / adc->getMaxValue(ADC_0)) -
settings.offset1) / (settings.convlow * 0.0001);

      if (abs((int16_t(value * 3300 / adc->getMaxValue(ADC_0)) -
settings.offset1)) <  settings.CurDead)
      {
        RawCur = 0;
      }
      if (debugCur != 0)
      {
        SERIALCONSOLE.print("  ");
        SERIALCONSOLE.print(int16_t(value * 3300 / adc-
>getMaxValue(ADC_0)) - settings.offset1);
        SERIALCONSOLE.print("  ");
        SERIALCONSOLE.print(RawCur);
        SERIALCONSOLE.print(" mA");
        SERIALCONSOLE.print("  ");
      }
    }
    else
    {
      if (debugCur != 0)
      {
        SERIALCONSOLE.println();
        SERIALCONSOLE.print("High Range: ");
        SERIALCONSOLE.print("Value ADC0: ");
      }
      value = (uint16_t)adc->analogReadContinuous(ADC_0); // the
unsigned is necessary for 16 bits, otherwise values larger than 3.3/2 V
are negative!
      if (debugCur != 0)
      {
        SERIALCONSOLE.print(value * 3300 / adc->getMaxValue(ADC_0)
);//- settings.offset2)
        SERIALCONSOLE.print("  ");
        SERIALCONSOLE.print(settings.offset2);
      }
      RawCur = int16_t((value * 3300 / adc->getMaxValue(ADC_0)) -
settings.offset2) / (settings.convhigh * 0.0001);
      if (value < 100 || value > (adc->getMaxValue(ADC_0) - 100))
      {
        RawCur = 0;
      }
      if (debugCur != 0)
      {
        SERIALCONSOLE.print("  ");
        SERIALCONSOLE.print((float(value * 3300 / adc-
>getMaxValue(ADC_0)) - settings.offset2));
        SERIALCONSOLE.print("  ");
        SERIALCONSOLE.print(RawCur);
        SERIALCONSOLE.print("mA");
        SERIALCONSOLE.print("  ");
      }
    }
  }
  if (settings.invertcur == 1)
  {
    RawCur = RawCur * -1;
  }
  lowpassFilter.input(RawCur);
  if (debugCur != 0)
  {
    SERIALCONSOLE.print(lowpassFilter.output());
  }
  currentact = lowpassFilter.output();
  if ( settings.cursens == Analoguedual)
  {
    if (sensor == 1)
    {
      if (currentact > 500 || currentact < -500 )
      {
        ampsecond = ampsecond + ((currentact * (millis() - lasttime) /
1000) / 1000);
        lasttime = millis();
      }
      else
      {
        lasttime = millis();
      }
    }
    if (sensor == 2)
    {
      if (currentact > 180000 || currentact < -18000 )
      {
        ampsecond = ampsecond + ((currentact * (millis() - lasttime) /
1000) / 1000);
        lasttime = millis();
      }
      else
      {
        lasttime = millis();
      }
    }
  }
  else
  {
    if (currentact > 500 || currentact < -500 )
    {
      ampsecond = ampsecond + ((currentact * (millis() - lasttime) /
1000) / 1000);
      lasttime = millis();
    }
    else
    {
      lasttime = millis();
    }
  }
  RawCur = 0;
}

void updateSOC()
{
  if (SOCset == 0)
  {
    if (millis() > 9000)
    {
      bms.setSensors(settings.IgnoreTemp, settings.IgnoreVolt);
    }
    if (millis() > 10000)
    {
      SOC = map(uint16_t(bms.getAvgCellVolt() * 1000),
settings.socvolt[0], settings.socvolt[2], settings.socvolt[1],
settings.socvolt[3]);

      ampsecond = (SOC * settings.CAP * settings.Pstrings * 10) /
0.27777777777778 ;
      SOCset = 1;
      SERIALCONSOLE.println("  ");
      SERIALCONSOLE.println("/////////////////////////////////////////
SOC SET /////////////////////////////////////////");
    }
  }
  if (settings.voltsoc == 1)
  {
    SOC = map(uint16_t(bms.getAvgCellVolt() * 1000),
settings.socvolt[0], settings.socvolt[2], settings.socvolt[1],
settings.socvolt[3]);

    ampsecond = (SOC * settings.CAP * settings.Pstrings * 10) /
0.27777777777778 ;
  }
  SOC = ((ampsecond * 0.27777777777778) / (settings.CAP *
settings.Pstrings * 1000)) * 100;
  if (SOC >= 100)
  {
    ampsecond = (settings.CAP * settings.Pstrings * 1000) /
0.27777777777778 ; //reset to full, dependant on given capacity. Need
to improve with auto correction for capcity.
    SOC = 100;
  }


  if (SOC < 0)
  {
    SOC = 0; //reset SOC this way the can messages remain in range for
other devices. Ampseconds will keep counting.
  }

  if (debug != 0)
  {
    if (settings.cursens == Analoguedual)
    {
      if (sensor == 1)
      {
        SERIALCONSOLE.print("Low Range ");
      }
      else
      {
        SERIALCONSOLE.print("High Range");
      }
    }
    if (settings.cursens == Analoguesing)
    {
      SERIALCONSOLE.print("Analogue Single ");
    }
    if (settings.cursens == Canbus)
    {
      SERIALCONSOLE.print("CANbus ");
    }
    SERIALCONSOLE.print("  ");
    SERIALCONSOLE.print(currentact);
    SERIALCONSOLE.print("mA");
    SERIALCONSOLE.print("  ");
    SERIALCONSOLE.print(SOC);
    SERIALCONSOLE.print("% SOC ");
    SERIALCONSOLE.print(ampsecond * 0.27777777777778, 2);
    SERIALCONSOLE.println ("mAh");
  }
}

void Prechargecon()
{
  if (digitalRead(IN1) == HIGH) //detect Key ON
  {
    digitalWrite(OUT4, HIGH);//Negative Contactor Close
    contctrl = 2;
    if (Pretimer +  settings.Pretime > millis() || currentact >
settings.Precurrent)
    {
      digitalWrite(OUT2, HIGH);//precharge
    }
    else //close main contactor
    {
      digitalWrite(OUT1, HIGH);//Positive Contactor Close
      contctrl = 3;
      bmsstatus = Drive;
      digitalWrite(OUT2, LOW);
    }
  }
  else
  {
    digitalWrite(OUT1, LOW);
    digitalWrite(OUT2, LOW);
    digitalWrite(OUT4, LOW);
    bmsstatus = Ready;
    contctrl = 0;
  }
}

void contcon()
{
  if (contctrl != contstat) //check for contactor request change
  {
    if ((contctrl & 1) == 0)
    {
      analogWrite(OUT5, 0);
      contstat = contstat & 254;
    }
    if ((contctrl & 2) == 0)
```

```
{
  analogWrite(OUT6, 0);
  contstat = contstat & 253;
}
if ((contctrl & 4) == 0)
{
  analogWrite(OUT7, 0);
  contstat = contstat & 251;
}
if ((contctrl & 1) == 1)
{
  if ((contstat & 1) != 1)
  {
    if (conttimer1 == 0)
    {
      analogWrite(OUT5, 255);
      conttimer1 = millis() + pulltime ;
    }
    if (conttimer1 < millis())
    {
      analogWrite(OUT5, settings.conthold);
      contstat = contstat | 1;
      conttimer1 = 0;
    }
  }
}

if ((contctrl & 2) == 2)
{
  if ((contstat & 2) != 2)
  {
    if (conttimer2 == 0)
    {
      Serial.println();
      Serial.println("pull in OUT6");
      analogWrite(OUT6, 255);
      conttimer2 = millis() + pulltime ;
    }
    if (conttimer2 < millis())
    {
      analogWrite(OUT6, settings.conthold);
      contstat = contstat | 2;
      conttimer2 = 0;
    }
  }
}
if ((contctrl & 4) == 4)
{
  if ((contstat & 4) != 4)
  {
    if (conttimer3 == 0)
    {
      Serial.println();
      Serial.println("pull in OUT7");
      analogWrite(OUT7, 255);
      conttimer3 = millis() + pulltime ;
    }
    if (conttimer3 < millis())
```

```
        {
          analogWrite(OUT7, settings.conthold);
          contstat = contstat | 4;
          conttimer3 = 0;
        }
      }
    }
    /*
       SERIALCONSOLE.print(conttimer);
       SERIALCONSOLE.print("  ");
       SERIALCONSOLE.print(contctrl);
       SERIALCONSOLE.print("  ");
       SERIALCONSOLE.print(contstat);
       SERIALCONSOLE.println("  ");
    */
  }
  if (contctrl == 0)
  {
    analogWrite(OUT5, 0);
    analogWrite(OUT6, 0);
    analogWrite(OUT7, 0);
  }
}

void calcur()
{
  adc->startContinuous(ACUR1, ADC_0);
  sensor = 1;
  SERIALCONSOLE.print(" Calibrating Current Offset ::::: ");
  while (x < 20)
  {
    offset1 = offset1 + ((uint16_t)adc->analogReadContinuous(ADC_0) *
3300 / adc->getMaxValue(ADC_0));
    SERIALCONSOLE.print(".");
    delay(100);
    x++;
  }
  offset1 = offset1 / 21;
  SERIALCONSOLE.print(offset1);
  SERIALCONSOLE.print(" current offset 1 calibrated ");
  SERIALCONSOLE.println("  ");
  x = 0;
  adc->startContinuous(ACUR2, ADC_0);
  sensor = 2;
  SERIALCONSOLE.print(" Calibrating Current Offset ::::: ");
  while (x < 20)
  {
    offset2 = offset2 + ((uint16_t)adc->analogReadContinuous(ADC_0) *
3300 / adc->getMaxValue(ADC_0));
    SERIALCONSOLE.print(".");
    delay(100);
    x++;
  }
  offset2 = offset2 / 21;
  SERIALCONSOLE.print(offset2);
  SERIALCONSOLE.print(" current offset 2 calibrated ");
  SERIALCONSOLE.println("  ");
}
```

```
void VEcan() //communication with Victron system over CAN
{
  msg.id  = 0x351;
  msg.len = 8;
  if (storagemode == 0)
  {
    msg.buf[0] = lowByte(uint16_t((settings.ChargeVsetpoint *
settings.Scells ) * 10));
    msg.buf[1] = highByte(uint16_t((settings.ChargeVsetpoint *
settings.Scells ) * 10));
  }
  else
  {
    msg.buf[0] = lowByte(uint16_t((settings.StoreVsetpoint *
settings.Scells ) * 10));
    msg.buf[1] = highByte(uint16_t((settings.StoreVsetpoint *
settings.Scells ) * 10));
  }
  msg.buf[2] = lowByte(chargecurrent);
  msg.buf[3] = highByte(chargecurrent);
  msg.buf[4] = lowByte(discurrent );
  msg.buf[5] = highByte(discurrent);
  msg.buf[6] = lowByte(uint16_t((settings.DischVsetpoint *
settings.Scells) * 10));
  msg.buf[7] = highByte(uint16_t((settings.DischVsetpoint *
settings.Scells) * 10));
  if (bmsstatus == Error)
  {
    msg.buf[2] = 0x00;
    msg.buf[3] = 0x00;
    msg.buf[4] = 0x00;
    msg.buf[5] = 0x00;
    alarm[2] = 0xF0;
  }
  else
  {
    alarm[2] = 0x00;
  }
  Can0.write(msg);

  msg.id  = 0x355;
  msg.len = 8;
  msg.buf[0] = lowByte(SOC);
  msg.buf[1] = highByte(SOC);
  msg.buf[2] = lowByte(SOH);
  msg.buf[3] = highByte(SOH);
  msg.buf[4] = lowByte(SOC * 10);
  msg.buf[5] = highByte(SOC * 10);
  msg.buf[6] = 0;
  msg.buf[7] = 0;
  Can0.write(msg);
  msg.id  = 0x356;
  msg.len = 8;
  msg.buf[0] = lowByte(uint16_t(bms.getPackVoltage() * 100));
  msg.buf[1] = highByte(uint16_t(bms.getPackVoltage() * 100));
  msg.buf[2] = lowByte(long(currentact / 100));
  msg.buf[3] = highByte(long(currentact / 100));
  msg.buf[4] = lowByte(uint16_t(bms.getAvgTemperature() * 10));
  msg.buf[5] = highByte(uint16_t(bms.getAvgTemperature() * 10));
  msg.buf[6] = 0;
  msg.buf[7] = 0;
  Can0.write(msg);
  delay(2);
  msg.id  = 0x35A;
  msg.len = 8;
  msg.buf[0] = alarm[0];//High temp  Low Voltage | High Voltage
  msg.buf[1] = alarm[1]; // High Discharge Current | Low Temperature
  msg.buf[2] = alarm[2]; //Internal Failure | High Charge current
  msg.buf[3] = alarm[3];// Cell Imbalance
  msg.buf[4] = warning[0];
  msg.buf[5] = warning[1];
  msg.buf[6] = warning[2];
  msg.buf[7] = warning[3];
  Can0.write(msg);
  msg.id  = 0x35E;
  msg.len = 8;
  msg.buf[0] = bmsname[0];
  msg.buf[1] = bmsname[1];
  msg.buf[2] = bmsname[2];
  msg.buf[3] = bmsname[3];
  msg.buf[4] = bmsname[4];
  msg.buf[5] = bmsname[5];
  msg.buf[6] = bmsname[6];
  msg.buf[7] = bmsname[7];
  Can0.write(msg);
  delay(2);
  msg.id  = 0x370;
  msg.len = 8;
  msg.buf[0] = bmsmanu[0];
  msg.buf[1] = bmsmanu[1];
  msg.buf[2] = bmsmanu[2];
  msg.buf[3] = bmsmanu[3];
  msg.buf[4] = bmsmanu[4];
  msg.buf[5] = bmsmanu[5];
  msg.buf[6] = bmsmanu[6];
  msg.buf[7] = bmsmanu[7];
  Can0.write(msg);
  if (balancecells == 1)
  {
    if (bms.getLowCellVolt() + settings.balanceHyst <
bms.getHighCellVolt())
    {
      msg.id  = 0x3c3;
      msg.len = 8;
      if (bms.getLowCellVolt() < settings.balanceVoltage)
      {
        msg.buf[0] = lowByte(uint16_t(settings.balanceVoltage * 1000));
        msg.buf[1] = highByte(uint16_t(settings.balanceVoltage *
1000));
      }
      else
      {
        msg.buf[0] = lowByte(uint16_t(bms.getLowCellVolt() * 1000));
        msg.buf[1] = highByte(uint16_t(bms.getLowCellVolt() * 1000));
      }
      msg.buf[2] =  0x01;
      msg.buf[3] =  0x04;
      msg.buf[4] =  0x03;
      msg.buf[5] =  0x00;
      msg.buf[6] =  0x00;
      msg.buf[7] = 0x00;
      Can0.write(msg);
    }
  }
}

void BMVmessage()//communication with the Victron Color Control System
over VEdirect
{
  lasttime = millis();
  x = 0;
  VE.write(13);
  VE.write(10);
  VE.write(myStrings[0]);
  VE.write(9);
  VE.print(bms.getPackVoltage() * 1000, 0);
  VE.write(13);
  VE.write(10);
  VE.write(myStrings[2]);
  VE.write(9);
  VE.print(currentact);
  VE.write(13);
  VE.write(10);
  VE.write(myStrings[4]);
  VE.write(9);
  VE.print(ampsecond * 0.27777777777778, 0); //consumed ah
  VE.write(13);
  VE.write(10);
  VE.write(myStrings[6]);
  VE.write(9);
  VE.print(SOC * 10); //SOC
  x = 8;
  while (x < 20)
  {
    VE.write(13);
    VE.write(10);
    VE.write(myStrings[x]);
    x ++;
    VE.write(9);
    VE.write(myStrings[x]);
    x ++;
  }
  VE.write(13);
  VE.write(10);
  VE.write("Checksum");
  VE.write(9);
  VE.write(0x50); //0x59
  delay(10);
  while (x < 44)
  {
    VE.write(13);
    VE.write(10);
    VE.write(myStrings[x]);
    x ++;
    VE.write(9);
    VE.write(myStrings[x]);
    x ++;
  }
  /*
    VE.write(13);
    VE.write(10);
    VE.write(myStrings[32]);
    VE.write(9);
    VE.print(bms.getLowVoltage()*1000,0);
    VE.write(13);
    VE.write(10);
    VE.write(myStrings[34]);
    VE.write(9);
    VE.print(bms.getHighVoltage()*1000,0);
    x=36;
    while(x < 43)
    {
     VE.write(13);
     VE.write(10);
     VE.write(myStrings[x]);
     x ++;
     VE.write(9);
     VE.write(myStrings[x]);
     x ++;
    }
  */
  VE.write(13);
  VE.write(10);
  VE.write("Checksum");
  VE.write(9);
  VE.write(231);
}
// Settings menu
void menu()
{
  incomingByte = Serial.read(); // read the incoming byte:
  if (menuload == 4)
  {
    switch (incomingByte)
    {
      case '1':
        menuload = 1;
        candebug = !candebug;
        incomingByte = 'd';
        break;
      case '2':
        menuload = 1;
        debugCur = !debugCur;
        incomingByte = 'd';
        break;
      case '3':
        menuload = 1;
        outputcheck = !outputcheck;
        if (outputcheck == 0)
        {
          contctrl = 0;
          digitalWrite(OUT1, LOW);
          digitalWrite(OUT2, LOW);
          digitalWrite(OUT3, LOW);
          digitalWrite(OUT4, LOW);
        }
        incomingByte = 'd';
        break;
      case '4':
        menuload = 1;
        inputcheck = !inputcheck;
        incomingByte = 'd';
        break;
      case '5':
        menuload = 1;
        settings.ESSmode = !settings.ESSmode;
        incomingByte = 'd';
        break;
      case '6':
        menuload = 1;
        cellspresent = bms.seriescells();
        incomingByte = 'd';
        break;
      case '7':
        menuload = 1;
        gaugedebug = !gaugedebug;
        incomingByte = 'd';
        break;
      case '8':
        menuload = 1;
        CSVdebug = !CSVdebug;
        incomingByte = 'd';
        break;
      case '9':
        menuload = 1;
        if (Serial.available() > 0)
        {
          debugdigits = Serial.parseInt();
        }
        if (debugdigits > 4)
        {
          debugdigits = 2;
        }
        incomingByte = 'd';
        break;
      case 113: //q for quite menu
        menuload = 0;
        incomingByte = 115;
        break;
      default:
        // if nothing else matches, do the default
        // default is optional
        break;
    }
  }
  if (menuload == 2)
  {
    switch (incomingByte)
    {
```

```
case 99: //c for calibrate zero offset
  calcur();
  break;
case '1':
  menuload = 1;
  settings.invertcur = !settings.invertcur;
  incomingByte = 'c';
  break;
case '2':
  menuload = 1;
  settings.voltsoc = !settings.voltsoc;
  incomingByte = 'c';
  break;
case '3':
  menuload = 1;
  if (Serial.available() > 0)
  {
    settings.ncur = Serial.parseInt();
  }
  menuload = 1;
  incomingByte = 'c';
  break;
case '4':
  menuload = 1;
  if (Serial.available() > 0)
  {
    settings.convlow = Serial.parseInt();
  }
  incomingByte = 'c';
  break;
case '5':
  menuload = 1;
  if (Serial.available() > 0)
  {
    settings.convhigh = Serial.parseInt();
  }
  incomingByte = 'c';
  break;
case '6':
  menuload = 1;
  if (Serial.available() > 0)
  {
    settings.CurDead = Serial.parseInt();
  }
  incomingByte = 'c';
  break;
case 113: //q for quite menu
  menuload = 0;
  incomingByte = 115;
  break;
case 115: //s for switch sensor
  settings.cursens ++;
  if (settings.cursens > 3)
  {
    settings.cursens = 0;
  }
  /*
    if (settings.cursens == Analoguedual)
```

```
        {
          settings.cursens = Canbus;
          SERIALCONSOLE.println("  ");
          SERIALCONSOLE.print(" CANbus Current Sensor ");
          SERIALCONSOLE.println("  ");
        }
        else
        {
          settings.cursens = Analoguedual;
          SERIALCONSOLE.println("  ");
          SERIALCONSOLE.print(" Analogue Current Sensor ");
          SERIALCONSOLE.println("  ");
        }
      */
      menuload = 1;
      incomingByte = 'c';
      break;
    default:
      // if nothing else matches, do the default
      // default is optional
      break;
  }
}
if (menuload == 8)
{
  switch (incomingByte)
  {
    case '1': //e dispaly settings
      if (Serial.available() > 0)
      {
        settings.IgnoreTemp = Serial.parseInt();
      }
      if (settings.IgnoreTemp > 2)
      {
        settings.IgnoreTemp = 0;
      }
      bms.setSensors(settings.IgnoreTemp, settings.IgnoreVolt);
      menuload = 1;
      incomingByte = 'i';
      break;
    case '2':
      if (Serial.available() > 0)
      {
        settings.IgnoreVolt = Serial.parseInt();
        settings.IgnoreVolt = settings.IgnoreVolt * 0.001;
        bms.setSensors(settings.IgnoreTemp, settings.IgnoreVolt);
        // Serial.println(settings.IgnoreVolt);
        menuload = 1;
        incomingByte = 'i';
      }
      break;
    case 113: //q to go back to main menu
      menuload = 0;
      incomingByte = 115;
      break;
  }
}
if (menuload == 7)
```

```
{
  switch (incomingByte)
  {
    case '1':
      if (Serial.available() > 0)
      {
        settings.WarnOff = Serial.parseInt();
        settings.WarnOff = settings.WarnOff * 0.001;
        menuload = 1;
        incomingByte = 'a';
      }
      break;
    case '2':
      if (Serial.available() > 0)
      {
        settings.CellGap = Serial.parseInt();
        settings.CellGap = settings.CellGap * 0.001;
        menuload = 1;
        incomingByte = 'a';
      }
      break;
    case '3':
      if (Serial.available() > 0)
      {
        settings.WarnToff = Serial.parseInt();
        menuload = 1;
        incomingByte = 'a';
      }
      break;
    case '4':
      if (Serial.available() > 0)
      {
        settings.UnderDur = Serial.parseInt();
        menuload = 1;
        incomingByte = 'a';
      }
      break;
    case 113: //q to go back to main menu
      menuload = 0;
      incomingByte = 115;
      break;
  }
}
if (menuload == 6) //Charging settings
{
  switch (incomingByte)
  {
    case 113: //q to go back to main menu

      menuload = 0;
      incomingByte = 115;
      break;
    case '1':
      if (Serial.available() > 0)
      {
        settings.ChargeVsetpoint = Serial.parseInt();
        settings.ChargeVsetpoint = settings.ChargeVsetpoint / 1000;
        menuload = 1;
        incomingByte = 'e';
      }
      break;
    case '2':
      if (Serial.available() > 0)
      {
        settings.ChargeHys = Serial.parseInt();
        settings.ChargeHys = settings.ChargeHys / 1000;
        menuload = 1;
        incomingByte = 'e';
      }
      break;
    case '4':
      if (Serial.available() > 0)
      {
        settings.chargecurrentend = Serial.parseInt() * 10;
        menuload = 1;
        incomingByte = 'e';
      }
      break;
    case '3':
      if (Serial.available() > 0)
      {
        settings.chargecurrentmax = Serial.parseInt() * 10;
        menuload = 1;
        incomingByte = 'e';
      }
      break;
    case '5': //1 Over Voltage Setpoint
      settings.chargertype = settings.chargertype + 1;
      if (settings.chargertype > 5)
      {
        settings.chargertype = 0;
      }
      menuload = 1;
      incomingByte = 'e';
      break;
    case '6':
      if (Serial.available() > 0)
      {
        settings.chargerspd = Serial.parseInt();
        menuload = 1;
        incomingByte = 'e';
      }
      break;
  }
}
if (menuload == 5)
{
  switch (incomingByte)
  {
    case '1':
      if (Serial.available() > 0)
      {
        settings.Pretime = Serial.parseInt();
        menuload = 1;
        incomingByte = 'k';
      }
      break;
    case '2':
      if (Serial.available() > 0)
      {
        settings.Precurrent = Serial.parseInt();
        menuload = 1;
        incomingByte = 'k';
      }
      break;
    case '3':
      if (Serial.available() > 0)
      {
        settings.conthold = Serial.parseInt();
        menuload = 1;
        incomingByte = 'k';
      }
      break;
    case '4':
      if (Serial.available() > 0)
      {
        settings.gaugelow = Serial.parseInt();
        gaugedebug = 2;
        gaugeupdate();
        menuload = 1;
        incomingByte = 'k';
      }
      break;
    case '5':
      if (Serial.available() > 0)
      {
        settings.gaugehigh = Serial.parseInt();
        gaugedebug = 3;
        gaugeupdate();
        menuload = 1;
        incomingByte = 'k';
      }
      break;
    case 113: //q to go back to main menu
      gaugedebug = 0;
      menuload = 0;
      incomingByte = 115;
      break;
  }
}
if (menuload == 3)
{
  switch (incomingByte)
  {
    case 113: //q to go back to main menu
      menuload = 0;
      incomingByte = 115;
      break;
    case 'f': //f factory settings
      loadSettings();
      SERIALCONSOLE.println("  ");
      SERIALCONSOLE.println("  ");
      SERIALCONSOLE.println("  ");
      SERIALCONSOLE.println(" Coded Settings Loaded ");
      SERIALCONSOLE.println("  ");
      menuload = 1;
      incomingByte = 'b';
      break;

    case 114: //r for reset
      SOCset = 0;
      SERIALCONSOLE.println("  ");
      SERIALCONSOLE.print(" mAh Reset ");
      SERIALCONSOLE.println("  ");
      menuload = 1;
      incomingByte = 'b';
      break;

    case '1': //1 Over Voltage Setpoint
      if (Serial.available() > 0)
      {
        settings.OverVSetpoint = Serial.parseInt();
        settings.OverVSetpoint = settings.OverVSetpoint / 1000;
        menuload = 1;
        incomingByte = 'b';
      }
      break;

    case 'g':
      if (Serial.available() > 0)
      {
        settings.StoreVsetpoint = Serial.parseInt();
        settings.StoreVsetpoint = settings.StoreVsetpoint / 1000;
        menuload = 1;
        incomingByte = 'b';
      }

    case 'b':
      if (Serial.available() > 0)
      {
        settings.socvolt[0] = Serial.parseInt();
        menuload = 1;
        incomingByte = 'b';
      }
      break;

    case 'c':
      if (Serial.available() > 0)
      {
        settings.socvolt[1] = Serial.parseInt();
        menuload = 1;
        incomingByte = 'b';
      }
      break;

    case 'd':
      if (Serial.available() > 0)
      {
        settings.socvolt[2] = Serial.parseInt();
        menuload = 1;
        incomingByte = 'b';
      }
      break;

    case 'e':
      if (Serial.available() > 0)
      {
        settings.socvolt[3] = Serial.parseInt();
        menuload = 1;
        incomingByte = 'b';
      }
      break;

    case '9': //Discharge Voltage Setpoint
      if (Serial.available() > 0)
      {
        settings.DischVsetpoint = Serial.parseInt();
        settings.DischVsetpoint = settings.DischVsetpoint / 1000;
        menuload = 1;
        incomingByte = 'b';
      }
      break;

    case '0': //c Pstrings
      if (Serial.available() > 0)
      {
        settings.Pstrings = Serial.parseInt();
        menuload = 1;
        incomingByte = 'b';
        bms.setPstrings(settings.Pstrings);
      }
      break;

    case 'a': //
      if (Serial.available() > 0)
      {
        settings.Scells  = Serial.parseInt();
        menuload = 1;
        incomingByte = 'b';
      }
      break;

    case '2': //2 Under Voltage Setpoint
      if (Serial.available() > 0)
      {
        settings.UnderVSetpoint = Serial.parseInt();
        settings.UnderVSetpoint =  settings.UnderVSetpoint / 1000;
        menuload = 1;
        incomingByte = 'b';
      }
      break;

    case '3': //3 Over Temperature Setpoint
      if (Serial.available() > 0)
      {
        settings.OverTSetpoint = Serial.parseInt();
        menuload = 1;
        incomingByte = 'b';
      }
      break;
    case '4': //4 Udner Temperature Setpoint
      if (Serial.available() > 0)
      {
        settings.UnderTSetpoint = Serial.parseInt();
        menuload = 1;
        incomingByte = 'b';
      }
      break;

    case '5': //5 Balance Voltage Setpoint
      if (Serial.available() > 0)
      {
        settings.balanceVoltage = Serial.parseInt();
        settings.balanceVoltage = settings.balanceVoltage / 1000;
        menuload = 1;
        incomingByte = 'b';
      }
      break;

    case '6': //6 Balance Voltage Hystersis
      if (Serial.available() > 0)
      {
        settings.balanceHyst = Serial.parseInt();
        settings.balanceHyst =  settings.balanceHyst / 1000;
        menuload = 1;
        incomingByte = 'b';
      }
      break;

    case '7'://7 Battery Capacity inAh
      if (Serial.available() > 0)
      {
        settings.CAP = Serial.parseInt();
        menuload = 1;
        incomingByte = 'b';
      }
      break;

    case '8':// discurrent in A
      if (Serial.available() > 0)
      {
        settings.discurrentmax = Serial.parseInt() * 10;
        menuload = 1;
        incomingByte = 'b';
      }
      break;
  }
}
if (menuload == 1)
{
  switch (incomingByte)
  {
    case 'R'://restart
      CPU_REBOOT ;
      break;

    case 'i': //Ignore Value Settings
```

```
      while (Serial.available()) {Serial.read();}
        SERIALCONSOLE.println();
        SERIALCONSOLE.println();
        SERIALCONSOLE.println();
        SERIALCONSOLE.println();
        SERIALCONSOLE.println();
        SERIALCONSOLE.println("Ignore Value Settings");
        SERIALCONSOLE.print("1 - Temp Sensor Setting:");
        SERIALCONSOLE.println(settings.IgnoreTemp);
        SERIALCONSOLE.print("2 - Voltage Under Which To Ignore
Cells:");
        SERIALCONSOLE.print(settings.IgnoreVolt * 1000, 0);
        SERIALCONSOLE.println("mV");
        SERIALCONSOLE.println("q - Go back to menu");
        menuload = 8;
        break;

      case 'e': //Charging settings
      while (Serial.available()) {Serial.read();}
        SERIALCONSOLE.println();
        SERIALCONSOLE.println();
        SERIALCONSOLE.println();
        SERIALCONSOLE.println();
        SERIALCONSOLE.println();
        SERIALCONSOLE.println("Charging Settings");
        SERIALCONSOLE.print("1 - Cell Charge Voltage Limit Setpoint:
");
        SERIALCONSOLE.print(settings.ChargeVsetpoint * 1000, 0);
        SERIALCONSOLE.println("mV");
        SERIALCONSOLE.print("2 - Charge Hystersis: ");
        SERIALCONSOLE.print(settings.ChargeHys * 1000, 0 );
        SERIALCONSOLE.println("mV");
        if (settings.chargertype > 0)
        {
          SERIALCONSOLE.print("3 - Pack Max Charge Current: ");
          SERIALCONSOLE.print(settings.chargecurrentmax * 0.1);
          SERIALCONSOLE.println("A");
          SERIALCONSOLE.print("4- Pack End of Charge Current: ");
          SERIALCONSOLE.print(settings.chargecurrentend * 0.1);
          SERIALCONSOLE.println("A");
        }
        SERIALCONSOLE.print("5- Charger Type: ");
        switch (settings.chargertype)
        {
          case 0:
            SERIALCONSOLE.print("Relay Control");
            break;
          case 1:
            SERIALCONSOLE.print("Brusa NLG5xx");
            break;
          case 2:
            SERIALCONSOLE.print("Volt Charger");
            break;
          case 3:
            SERIALCONSOLE.print("Eltek Charger");
            break;
          case 4:
            SERIALCONSOLE.print("Elcon Charger");
            break;
          case 5:
            SERIALCONSOLE.print("Victron Charger");
            break;
        }
        SERIALCONSOLE.println();
        if (settings.chargertype > 0)
        {
          SERIALCONSOLE.print("6- Charger Can Msg Spd: ");
          SERIALCONSOLE.print(settings.chargerspd);
          SERIALCONSOLE.println("mS");
        }
        /*
          SERIALCONSOLE.print("7- Can Speed:");
          SERIALCONSOLE.print(settings.canSpeed/1000);
          SERIALCONSOLE.println("kbps");
        */
        SERIALCONSOLE.println();
        SERIALCONSOLE.println("q - Go back to menu");
        menuload = 6;
        break;

      case 'a': //Alarm and Warning settings
      while (Serial.available()) {Serial.read();}
        SERIALCONSOLE.println();
        SERIALCONSOLE.println();
        SERIALCONSOLE.println();
        SERIALCONSOLE.println();
        SERIALCONSOLE.println();
        SERIALCONSOLE.println("Alarm and Warning Settings Menu");
        SERIALCONSOLE.print("1 - Voltage Warning Offset: ");
        SERIALCONSOLE.print(settings.WarnOff * 1000, 0);
        SERIALCONSOLE.println("mV");
        SERIALCONSOLE.print("2 - Cell Voltage Difference Alarm: ");
        SERIALCONSOLE.print(settings.CellGap * 1000, 0);
        SERIALCONSOLE.println("mV");
        SERIALCONSOLE.print("3 - Temp Warning Offset: ");
        SERIALCONSOLE.print(settings.WarnToff);
        SERIALCONSOLE.println(" C");
        SERIALCONSOLE.print("4 - Temp Warning Offset: ");
        SERIALCONSOLE.print(settings.UnderDur);
        SERIALCONSOLE.println(" mS");
        menuload = 7;
        break;

      case 'k': //contactor settings
      while (Serial.available()) {Serial.read();}
        SERIALCONSOLE.println();
        SERIALCONSOLE.println();
        SERIALCONSOLE.println();
        SERIALCONSOLE.println();
        SERIALCONSOLE.println();
        SERIALCONSOLE.println("Contactor and Gauge Settings Menu");
        SERIALCONSOLE.print("1 - PreCharge Timer: ");
        SERIALCONSOLE.print(settings.Pretime);
        SERIALCONSOLE.println("mS");
        SERIALCONSOLE.print("2 - PreCharge Finish Current: ");
        SERIALCONSOLE.print(settings.Precurrent);
```

```
    SERIALCONSOLE.println(" mA");
    SERIALCONSOLE.print("3 - PWM contactor Hold 0-255 :");
    SERIALCONSOLE.println(settings.conthold);
    SERIALCONSOLE.print("4 - PWM for Gauge Low 0-255 :");
    SERIALCONSOLE.println(settings.gaugelow);
    SERIALCONSOLE.print("5 - PWM for Gauge High 0-255 :");
    SERIALCONSOLE.println(settings.gaugehigh);
    menuload = 5;
    break;

  case 113: //q to go back to main menu
    EEPROM.put(0, settings); //save all change to eeprom
    menuload = 0;
    debug = 1;
    break;
  case 'd': //d for debug settings
    while (Serial.available()) {Serial.read();}
    SERIALCONSOLE.println();
    SERIALCONSOLE.println();
    SERIALCONSOLE.println();
    SERIALCONSOLE.println();
    SERIALCONSOLE.println();
    SERIALCONSOLE.println("Debug Settings Menu");
    SERIALCONSOLE.println("Toggle on/off");
    SERIALCONSOLE.print("1 - Can Debug :");
    SERIALCONSOLE.println(candebug);
    SERIALCONSOLE.print("2 - Current Debug :");
    SERIALCONSOLE.println(debugCur);
    SERIALCONSOLE.print("3 - Output Check :");
    SERIALCONSOLE.println(outputcheck);
    SERIALCONSOLE.print("4 - Input Check :");
    SERIALCONSOLE.println(inputcheck);
    SERIALCONSOLE.print("5 - ESS mode :");
    SERIALCONSOLE.println(settings.ESSmode);
    SERIALCONSOLE.print("6 - Cells Present Reset :");
    SERIALCONSOLE.println(cellspresent);
    SERIALCONSOLE.print("7 - Gauge Debug :");
    SERIALCONSOLE.println(gaugedebug);
    SERIALCONSOLE.print("8 - CSV Output :");
    SERIALCONSOLE.println(CSVdebug);
    SERIALCONSOLE.print("9 - Decimal Places to Show :");
    SERIALCONSOLE.println(debugdigits);
    SERIALCONSOLE.println("q - Go back to menu");
    menuload = 4;
    break;

  case 99: //c for calibrate zero offset
  while (Serial.available()) {Serial.read();}
    SERIALCONSOLE.println();
    SERIALCONSOLE.println();
    SERIALCONSOLE.println();
    SERIALCONSOLE.println();
    SERIALCONSOLE.println();
    SERIALCONSOLE.println("Current Sensor Calibration Menu");
    SERIALCONSOLE.println("c - To calibrate sensor offset");
    SERIALCONSOLE.print("s - Current Sensor Type : ");
    switch (settings.cursens)
    {
          case Analoguedual:
            SERIALCONSOLE.println(" Analogue Dual Current Sensor ");
            break;
          case Analoguesing:
            SERIALCONSOLE.println(" Analogue Single Current Sensor ");
            break;
          case Canbus:
            SERIALCONSOLE.println(" Canbus Current Sensor ");
            break;
          default:
            SERIALCONSOLE.println("Undefined");
            break;
        }
        SERIALCONSOLE.print("1 - invert current :");
        SERIALCONSOLE.println(settings.invertcur);
        SERIALCONSOLE.print("2 - Pure Voltage based SOC :");
        SERIALCONSOLE.println(settings.voltsoc);
        SERIALCONSOLE.print("3 - Current Multiplication :");
        SERIALCONSOLE.println(settings.ncur);
        if (settings.cursens == Analoguesing || settings.cursens ==
Analoguedual)
        {
          SERIALCONSOLE.print("4 - Analogue Low Range Conv:");
          SERIALCONSOLE.print(settings.convlow * 0.1, 1);
          SERIALCONSOLE.println(" mV/A");
        }
        if ( settings.cursens == Analoguedual)
        {
          SERIALCONSOLE.print("5 - Analogue High Range Conv:");
          SERIALCONSOLE.print(settings.convhigh * 0.1, 1);
          SERIALCONSOLE.println(" mV/A");
        }
        if (settings.cursens == Analoguesing || settings.cursens ==
Analoguedual)
        {
          SERIALCONSOLE.print("6 - Current Sensor Deadband:");
          SERIALCONSOLE.print(settings.CurDead);
          SERIALCONSOLE.println(" mV");
        }
        SERIALCONSOLE.println("q - Go back to menu");
        menuload = 2;
        break;

      case 98: //c for calibrate zero offset
      while (Serial.available()) {Serial.read();}
        SERIALCONSOLE.println();
        SERIALCONSOLE.println();
        SERIALCONSOLE.println();
        SERIALCONSOLE.println();
        SERIALCONSOLE.println();
        SERIALCONSOLE.println("Battery Settings Menu");
        SERIALCONSOLE.println("r - Reset AH counter");
        SERIALCONSOLE.println("f - Reset to Coded Settings");
        SERIALCONSOLE.println("q - Go back to menu");
        SERIALCONSOLE.println();
        SERIALCONSOLE.println();
        SERIALCONSOLE.print("1 - Cell Over Voltage Setpoint: ");
        SERIALCONSOLE.print(settings.OverVSetpoint * 1000, 0);
        SERIALCONSOLE.print("mV");
        SERIALCONSOLE.println("  ");
        SERIALCONSOLE.print("2 - Cell Under Voltage Setpoint: ");
        SERIALCONSOLE.print(settings.UnderVSetpoint * 1000, 0);
        SERIALCONSOLE.print("mV");
        SERIALCONSOLE.println("  ");
        SERIALCONSOLE.print("3 - Over Temperature Setpoint: ");
        SERIALCONSOLE.print(settings.OverTSetpoint);
        SERIALCONSOLE.print("C");
        SERIALCONSOLE.println("  ");
        SERIALCONSOLE.print("4 - Under Temperature Setpoint: ");
        SERIALCONSOLE.print(settings.UnderTSetpoint);
        SERIALCONSOLE.print("C");
        SERIALCONSOLE.println("  ");
        SERIALCONSOLE.print("5 - Cell Balance Voltage Setpoint: ");
        SERIALCONSOLE.print(settings.balanceVoltage * 1000, 0);
        SERIALCONSOLE.print("mV");
        SERIALCONSOLE.println("  ");
        SERIALCONSOLE.print("6 - Balance Voltage Hystersis: ");
        SERIALCONSOLE.print(settings.balanceHyst * 1000, 0);
        SERIALCONSOLE.print("mV");
        SERIALCONSOLE.println("  ");
        SERIALCONSOLE.print("7 - Ah Battery Capacity: ");
        SERIALCONSOLE.print(settings.CAP);
        SERIALCONSOLE.print("Ah");
        SERIALCONSOLE.println("  ");
        SERIALCONSOLE.print("8 - Pack Max Discharge: ");
        SERIALCONSOLE.print(settings.discurrentmax * 0.1);
        SERIALCONSOLE.print("A");
        SERIALCONSOLE.println("  ");
        SERIALCONSOLE.print("9 - Cell Discharge Voltage Limit Setpoint: 
");
        SERIALCONSOLE.print(settings.DischVsetpoint * 1000, 0);
        SERIALCONSOLE.print("mV");
        SERIALCONSOLE.println("  ");
        SERIALCONSOLE.print("0 - Slave strings in parallel: ");
        SERIALCONSOLE.print(settings.Pstrings);
        SERIALCONSOLE.println("  ");
        SERIALCONSOLE.print("a - Cells in Series per String: ");
        SERIALCONSOLE.print(settings.Scells );
        SERIALCONSOLE.println("  ");
        SERIALCONSOLE.print("b - setpoint 1: ");
        SERIALCONSOLE.print(settings.socvolt[0] );
        SERIALCONSOLE.print("mV");
        SERIALCONSOLE.println("  ");
        SERIALCONSOLE.print("c - SOC setpoint 1:");
        SERIALCONSOLE.print(settings.socvolt[1] );
        SERIALCONSOLE.print("%");
        SERIALCONSOLE.println("  ");
        SERIALCONSOLE.print("d - setpoint 2: ");
        SERIALCONSOLE.print(settings.socvolt[2] );
        SERIALCONSOLE.print("mV");
        SERIALCONSOLE.println("  ");
        SERIALCONSOLE.print("e - SOC setpoint 2: ");
        SERIALCONSOLE.print(settings.socvolt[3] );
        SERIALCONSOLE.print("%");
        SERIALCONSOLE.println("  ");
        SERIALCONSOLE.print("g - Storage Setpoint: ");
        SERIALCONSOLE.print(settings.StoreVsetpoint * 1000, 0 );
        SERIALCONSOLE.print("mV");
        SERIALCONSOLE.println("  ");
        SERIALCONSOLE.println();
        menuload = 3;
        break;
      default:
        // if nothing else matches, do the default
        // default is optional
        break;
    }
  }
  if (incomingByte == 115 & menuload == 0)
  {
    while (Serial.available()) {Serial.read();}
    SERIALCONSOLE.println();
    SERIALCONSOLE.println("MENU");
    SERIALCONSOLE.println("Debugging Paused");
    SERIALCONSOLE.print("Firmware Version : ");
    SERIALCONSOLE.println(firmver);
    SERIALCONSOLE.println("b - Battery Settings");
    SERIALCONSOLE.println("a - Alarm and Warning Settings");
    SERIALCONSOLE.println("e - Charging Settings");
    SERIALCONSOLE.println("c - Current Sensor Calibration");
    SERIALCONSOLE.println("k - Contactor and Gauge Settings");
    SERIALCONSOLE.println("i - Ignore Value Settings");
    SERIALCONSOLE.println("d - Debug Settings");
    SERIALCONSOLE.println("R - Restart BMS");
    SERIALCONSOLE.println("q - exit menu");
    debug = 0;
    menuload = 1;
  }
}
void canread()
{
  Can0.read(inMsg);
  // Read data: len = data length, buf = data byte(s)
  if (inMsg.id == 0x3c)
  {
    CAB300();
  }
  if (inMsg.id < 0x300)//do VW BMS magic if ids are ones identified to
be modules
  {
    if (candebug == 1)
    {
      bms.decodecan(inMsg, 1); //do VW BMS if ids are ones identified
to be modules
    }
    else
    {
      bms.decodecan(inMsg, 0); //do VW BMS if ids are ones identified
to be modules
    }
  }
  if ((inMsg.id & 0x1FFFFFFF) < 0x1A5554F0 && (inMsg.id & 0x1FFFFFFF) >
0x1A555400)   // Determine if ID is Temperature CAN-ID
  {
    if (candebug == 1)
    {
      bms.decodetemp(inMsg, 1);

    }
    else
    {
      bms.decodetemp(inMsg, 0);
    }
  }
  if (candebug == 1)
  {
    Serial.print(millis());
    if ((inMsg.id & 0x80000000) == 0x80000000)    // Determine if ID is
standard (11 bits) or extended (29 bits)
      sprintf(msgString, "Extended ID: 0x%.8lX  DLC: %1d  Data:",
(inMsg.id & 0x1FFFFFFF), inMsg.len);
    else
      sprintf(msgString, ",0x%.3lX,false,%1d", inMsg.id, inMsg.len);

    Serial.print(msgString);

    if ((inMsg.id & 0x40000000) == 0x40000000) {  // Determine if
message is a remote request frame.
      sprintf(msgString, " REMOTE REQUEST FRAME");
      Serial.print(msgString);
    } else {
      for (byte i = 0; i < inMsg.len; i++) {
        sprintf(msgString, ", 0x%.2X", inMsg.buf[i]);
        Serial.print(msgString);
      }
    }
    Serial.println();
  }
}
void CAB300()
{
  for (int i = 0; i < 4; i++)
  {
    inbox = (inbox << 8) | inMsg.buf[i];
  }
  CANmilliamps = inbox;
  if (CANmilliamps > 0x80000000)
  {
    CANmilliamps -= 0x80000000;
  }
  else
  {
    CANmilliamps = (0x80000000 - CANmilliamps) * -1;
  }
  if (settings.cursens == Canbus)
  {
    RawCur = CANmilliamps;
    getcurrent();
  }
  if (candebug == 1)
  {
    Serial.println();
    Serial.print(CANmilliamps);
    Serial.print("mA ");
  }
}
void currentlimit()
{
  if (bmsstatus == Error)
  {
    discurrent = 0;
    chargecurrent = 0;
  }
  else
  {
    if (bms.getAvgTemperature() < settings.UnderTSetpoint)
    {
      discurrent = 0;
      chargecurrent = 0;
    }
    else
    {
      if (bms.getAvgTemperature() < settings.ChargeTSetpoint)
      {
        discurrent = settings.discurrentmax;
        chargecurrent = map(bms.getAvgTemperature(),
settings.UnderTSetpoint, settings.ChargeTSetpoint, 0,
settings.chargecurrentmax);
      }
      else
      {
        if (bms.getAvgTemperature() < settings.DisTSetpoint)
        {
          discurrent = settings.discurrentmax;
          chargecurrent = settings.chargecurrentmax;
        }
        else
        {
          if (bms.getAvgTemperature() < settings.OverTSetpoint)
          {
            discurrent = map(bms.getAvgTemperature(),
settings.DisTSetpoint, settings.OverTSetpoint, settings.discurrentmax,
0);
            chargecurrent = settings.chargecurrentmax;
          }
          else
          {
            discurrent = 0;
            chargecurrent = 0;
          }
        }
      }
    }
  }
  ///voltage influence on current///
  if (storagemode == 1)
  {
    if (bms.getHighCellVolt() > (settings.StoreVsetpoint -
settings.ChargeHys))
    {
      chargecurrent = map(bms.getHighCellVolt(),
(settings.StoreVsetpoint - settings.ChargeHys),
settings.StoreVsetpoint, settings.chargecurrentmax,
settings.chargecurrentend);
    }
    if (bms.getHighCellVolt() > settings.OverVSetpoint)
    {
      chargecurrent = 0;
    }
  }
  else
  {
    if (bms.getHighCellVolt() > (settings.ChargeVsetpoint -
settings.ChargeHys))
    {
      chargecurrent = map(bms.getHighCellVolt(),
(settings.ChargeVsetpoint - settings.ChargeHys),
settings.ChargeVsetpoint, settings.chargecurrentmax,
settings.chargecurrentend);
    }
    if (bms.getHighCellVolt() > settings.OverVSetpoint)
    {
      chargecurrent = 0;
    }
  }
  if (bms.getLowCellVolt() < settings.UnderVSetpoint ||
bms.getLowCellVolt() < settings.DischVsetpoint)
  {
    discurrent = 0;
  }
  ///No negative currents///
  if (discurrent < 0)
  {
    discurrent = 0;
  }
  if (chargecurrent < 0)
  {
    chargecurrent = 0;
  }
}
void inputdebug()
{
  Serial.println();
  Serial.print("Input: ");
  if (digitalRead(IN1))
  {
    Serial.print("1 ON  ");
  }
  else
  {
    Serial.print("1 OFF ");
  }
  if (digitalRead(IN3))
  {
    Serial.print("2 ON  ");
  }
  else
  {
    Serial.print("2 OFF ");
  }
  if (digitalRead(IN3))
  {
    Serial.print("3 ON  ");
  }
  else
  {
    Serial.print("3 OFF ");
  }
  if (digitalRead(IN4))
  {
    Serial.print("4 ON  ");
  }
  else
  {
    Serial.print("4 OFF ");
  }
  Serial.println();
}
void outputdebug()
{
  if (outputstate < 5)
  {
    digitalWrite(OUT1, HIGH);
    digitalWrite(OUT2, HIGH);
    digitalWrite(OUT3, HIGH);
    digitalWrite(OUT4, HIGH);
    analogWrite(OUT5, 255);
    analogWrite(OUT6, 255);
    analogWrite(OUT7, 255);
    analogWrite(OUT8, 255);
    outputstate ++;
  }
  else
  {
    digitalWrite(OUT1, LOW);
    digitalWrite(OUT2, LOW);
    digitalWrite(OUT3, LOW);
    digitalWrite(OUT4, LOW);
    analogWrite(OUT5, 0);
    analogWrite(OUT6, 0);
    analogWrite(OUT7, 0);
    analogWrite(OUT8, 0);
    outputstate ++;
  }
  if (outputstate > 10)
  {
    outputstate = 0;
  }
}
void sendcommand()
{
  msg.id  = controlid;
  msg.len = 8;
  msg.buf[0] = 0x00;
  msg.buf[1] = 0x00;
  msg.buf[2] = 0x00;
  msg.buf[3] = 0x00;
  msg.buf[4] = 0x00;
  msg.buf[5] = 0x00;
  msg.buf[6] = 0x00;
  msg.buf[7] = 0x00;
  Can0.write(msg);
  delay(1);
  msg.id  = controlid;
  msg.len = 8;
  msg.buf[0] = 0x45;
  msg.buf[1] = 0x01;
  msg.buf[2] = 0x28;
  msg.buf[3] = 0x00;
  msg.buf[4] = 0x00;
  msg.buf[5] = 0x00;
  msg.buf[6] = 0x00;
  msg.buf[7] = 0x30;
  Can0.write(msg);
}
void resetwdog()
{
  noInterrupts();                                     //   No - reset
WDT
  WDOG_REFRESH = 0xA602;
  WDOG_REFRESH = 0xB480;
  interrupts();
}
void pwmcomms()
{
  int p = 0;
  p = map((currentact * 0.001), pwmcurmin, pwmcurmax, 50 , 255);
  analogWrite(OUT7, p);
  /*
    Serial.println();
      Serial.print(p*100/255);
      Serial.print(" OUT8 ");
  */
  if (bms.getLowCellVolt() < settings.UnderVSetpoint)
  {
    analogWrite(OUT8, 224); //12V to 10V converter 1.5V
  }
  else
  {
    p = map(SOC, 0, 100, 220, 50);
    analogWrite(OUT8, p); //2V to 10V converter 1.5-10V
  }
  /*
      Serial.println();
      Serial.print(p*100/255);
      Serial.print(" OUT7 ");
  */
}
void Serialexp()
{
  /*
    incomingByte = SERIALBMS.read(); // read the incoming byte:
    Serial.println();
    Serial.print(incomingByte);
    Serial.print("|");
    incomingByte = SERIALBMS.read(); // read the incoming byte:
    if (incomingByte == 0xFF)
    {
    Serial.println();
    Serial.print(incomingByte);
    incomingByte = SERIALBMS.read(); // read the incoming byte:
    Serial.print("|");
    Serial.print(incomingByte);
    Serial.print("|");
    Serial.print(incomingByte);
    if (settings.Serialexp == 1) //Do Serial Master Things
    {
    Serial.print(SERIALBMS.read(), HEX);
    Serial.print("|");
    Serial.print(SERIALBMS.read(), HEX);
    }
    if (settings.Serialexp == 2) //Do Serial Slave Things
    {
    switch (incomingByte)
    {
    case 0x00: //q to go back to main menu
    if (SerialID == 0)
    {
      SerialID = SERIALBMS.read();
      SERIALBMS.write(0x01); //response is 1 higher than sent id
      SERIALBMS.write(SerialID);
      Serial.print("New ID : ");
      Serial.print(SerialID);
    }
    else
    {
      SERIALBMS.write(0xFF);
      SERIALBMS.write(0x00);
      SERIALBMS.write(SERIALBMS.read());
    }
    break;
    }
    }
    }
  */
}
void SerialReqData()
{
  /*
    SERIALBMS.write(0x12);
  */
}
void Serialslaveinit()
{
  /*
    int buff[8];
    while (1 == 1)
    {
    for (int I = 1; I < 51; I++)
    {
      SERIALBMS.write(0xFF);
      SERIALBMS.write(0x00);
      SERIALBMS.write(I);
      Serial.write(" | ");
      delay(2);
      if (SERIALBMS.available() > 0)
      {
        for (int x = 0; x < 4; x++)
        {
          buff[x] = SERIALBMS.read();
          Serial.write(buff[0]);
          Serial.write(buff[1]);
          Serial.write(buff[2]);
        }
        if (buff[0] = 0xFF)
        {
          if (buff[1] == I)
          {
            break;
          }
        }
      }
      else
      {
        Serial.write("No Serial Slaves Found");
        break;
      }
    }
    break;
    }
  */
}
void chargercomms()
{
  if (settings.chargertype == Elcon)
  {
    msg.id  =  0x1806E5F4; //broadcast to all Elteks
    msg.len = 8;
    msg.ext = 1;
    msg.buf[0] = highByte(uint16_t(settings.ChargeVsetpoint *
settings.Scells * 10));
    msg.buf[1] = lowByte(uint16_t(settings.ChargeVsetpoint *
settings.Scells * 10));
    msg.buf[2] = highByte(chargecurrent / ncharger);
    msg.buf[3] = lowByte(chargecurrent / ncharger);
    msg.buf[4] = 0x00;
    msg.buf[5] = 0x00;
    msg.buf[6] = 0x00;
    msg.buf[7] = 0x00;

    Can0.write(msg);
    msg.ext = 0;
  }
  if (settings.chargertype == Eltek)
  {
    msg.id  = 0x2FF; //broadcast to all Elteks
    msg.len = 7;
    msg.buf[0] = 0x01;
    msg.buf[1] = lowByte(1000);
    msg.buf[2] = highByte(1000);
    msg.buf[3] = lowByte(uint16_t(settings.ChargeVsetpoint *
settings.Scells * 10));
    msg.buf[4] = highByte(uint16_t(settings.ChargeVsetpoint *
settings.Scells * 10));
    msg.buf[5] = lowByte(chargecurrent / ncharger);
    msg.buf[6] = highByte(chargecurrent / ncharger);
    Can0.write(msg);
  }
  if (settings.chargertype == BrusaNLG5)
  {
    msg.id  = chargerid1;
    msg.len = 7;
    msg.buf[0] = 0x80;
    /*
      if (chargertoggle == 0)
      {
      msg.buf[0] = 0x80;
      chargertoggle++;
      }
      else
      {
      msg.buf[0] = 0xC0;
      chargertoggle = 0;
      }
    */
    if (digitalRead(IN2) == LOW)//Gen OFF
    {
      msg.buf[1] = highByte(maxac1 * 10);
      msg.buf[2] = lowByte(maxac1 * 10);
    }
    else
    {
      msg.buf[1] = highByte(maxac2 * 10);
      msg.buf[2] = lowByte(maxac2 * 10);
    }
    msg.buf[5] = highByte(chargecurrent / ncharger);
    msg.buf[6] = lowByte(chargecurrent / ncharger);
    msg.buf[3] = highByte(uint16_t(((settings.ChargeVsetpoint *
settings.Scells ) - chargerendbulk) * 10));
    msg.buf[4] = lowByte(uint16_t(((settings.ChargeVsetpoint *
settings.Scells ) - chargerendbulk)  * 10));
    Can0.write(msg);
    delay(2);
    msg.id  = chargerid2;
    msg.len = 7;
    msg.buf[0] = 0x80;
    if (digitalRead(IN2) == LOW)//Gen OFF
    {
      msg.buf[1] = highByte(maxac1 * 10);
      msg.buf[2] = lowByte(maxac1 * 10);
    }
    else
    {
      msg.buf[1] = highByte(maxac2 * 10);
      msg.buf[2] = lowByte(maxac2 * 10);
    }
    msg.buf[3] = highByte(uint16_t(((settings.ChargeVsetpoint *
settings.Scells ) - chargerend) * 10));
```

```
    msg.buf[4] = lowByte(uint16_t(((settings.ChargeVsetpoint *
settings.Scells ) - chargerend) * 10));
    msg.buf[5] = highByte(chargecurrent / ncharger);
    msg.buf[6] = lowByte(chargecurrent / ncharger);
    Can0.write(msg);
  }
  if (settings.chargertype == ChevyVolt)
  {
    msg.id  = 0x30E;
    msg.len = 1;
    msg.buf[0] = 0x02; //only HV charging , 0x03 hv and 12V charging
    Can0.write(msg);
    msg.id  = 0x304;
    msg.len = 4;
    msg.buf[0] = 0x40; //fixed
    if ((chargecurrent * 2) > 255)
    {
      msg.buf[1] = 255;
    }
    else
    {
      msg.buf[1] = (chargecurrent * 2);
    }
    if ((settings.ChargeVsetpoint * settings.Scells ) > 200)
    {
      msg.buf[2] = highByte(uint16_t((settings.ChargeVsetpoint *
settings.Scells ) * 2));
      msg.buf[3] = lowByte(uint16_t((settings.ChargeVsetpoint *
settings.Scells ) * 2));
    }
    else
    {
      msg.buf[2] = highByte( 400);
      msg.buf[3] = lowByte( 400);
    }
    Can0.write(msg);
  }
```