Atte Marttinen

Design and implementation of graphical user interface for electric motorcycle

Bachelor's thesis

Bachelor of electrical and automation engineering

2022

Kaakkois-Suomen ammattikorkeakoulu

Kaakkois-Suomen ammattikorkeakoulu

Degree	*Bachelor of Engineering*
Author (authors)	Atte Marttinen
Thesis title	Design and implementation of a graphical user interface for electric motorcycle
Commissioned by	Xamk
Time	2022
Pages	38 pages, 4 pages of appendices
Supervisor	Teemu Manninen

ABSTRACT

This thesis describes the designing and implementation process of a graphical user interface for electric motorcycle using Qt. The project was started at the end of year 2019 and finished in April of 2022. The project was carried out by three electrical and automation engineering students.

The motorcycle has numerous electronic devices whose information and state need to be displayed to the driver. The goal of this thesis was to create a user interface that is easy to understand and view while driving. The user interface was decided to be programmed by me because there were no good options available for this project in the market.

This thesis starts by introducing the main components of the motorcycle such as the battery management system and the motor controller. These components were chosen to be discussed in this thesis because an user interface developer must understand at least the basics of the system to be able to create a working interface for the vehicle.

The thesis then continues to discuss about CAN-bus protocol since it was used in this project as a communication method between the electronics of the motorcycle.

After the hardware of the system was described, the thesis proceeds to the actual implementation process of the project. At this point, the steps for setting the developing environment up are introduced. The software was developed with an open-source principle which means that one can use the software as desired. The source code can be used also as a template for any vehicle or system that uses CAN-bus as its communication method.

The outcome of this thesis was a working user interface that can display the information of the system to the driver. The modular design of the system simplifies the possible future developing work, meaning that components that use CAN-bus communication can be easily added to the system.

Keywords: CAN-bus, Qt, user interface, Raspberry Pi, electric motorcycle

South-Eastern Finland University of Applied Sciences

Tutkintonimike	Insinööri (AMK)
Tekijä/Tekijät	Atte Marttinen
Työn nimi	Sähkömoottoripyörän käyttöliittymän suunnittelu ja toteutus
Toimeksiantaja	Xamk
Vuosi	2022
Sivut	38 sivua, liitteitä 4 sivua
Työn ohjaaja(t)	Teemu Manninen

Tiivistelmä

Tässä opinnäytetyössä käydään läpi sähkömoottoripyörämuunnoksen graafisen käyttöliittymän suunnittelu- ja toteutusprosessia. Projekti alkoi vuoden 2019 lopulla ja päättyi pääsiäisenä 2022. Projektin toteutti kolme sähkö- ja automaatioinsinööriopiskelijaa. Projektitiimi valittiin kilpailulla, jonka tarkoituksena oli tehdä esisuunnitelma sähkömoottoripyörämuunnoksesta.

Moottoripyörässä on monia elektronisia laitteita, joiden tilasta täytyy saada tieto kuljettajalle. Tämän opinnäytetyön tarkoitus oli luoda helposti ymmärrettävä käyttöliittymä, jota pystyttäisiin lukemaan myös ajaessa. Käyttöliittymä päätettiin luoda itse, koska markkinoilla ei ollut sopivia näyttöjä tähän projektiin.

Opinnäytetyössä esitellään moottoripyörän oleellisimmat komponentit pikaisesti, jotta lukija ymmärtää niiden toiminnan korkealla tasolla. Näiden komponenttien toiminnan ymmärrys on suotavaa, jotta kehittäjä osaa suunnitella käyttöliittymän siten, että siitä pystyy näkemään laitteen tärkeimmät arvot, esimerkiksi akuston jännitteen.

CAN-väylän ohjelmointi ja kytkennät olivat iso osa tätä työtä, sillä moottoripyörän elektroniikka kommunikoi keskenään käyttäen tätä tiedonsiirtomuotoa.

Käyttöliittymä ohjelmointiin open-source-periaatteella, mikä tarkoittaa sitä, että kuka vain voi käyttää ohjelmaa haluamallaan tavalla. Opinnäytetyö sisältää tärkeimmät vaiheet ohjelmointiympäristön pystyttämiseen sekä ohjelmointiesimerkkejä. Yhdessä lähdekoodin sekä ohjelmointiesimerkkien kanssa lukija pystyy hahmottamaan ohjelman toiminnan ja kehittämään sitä niin halutessaan.

Työn tuloksena syntyi tavoitteet saavuttanut käyttöliittymä, joka osoittautui testien perusteella toimivaksi. CAN-väylän toiminta oli luotettavaa, eikä ohjelma kaatunut bugeihin. Modulaarinen suunnittelu helpottaa mahdollisen jatkokehityksen, sillä ainoa vaatimus pyörään asennettavalle elektroniikalle on, että sen täytyy kyetä kommunikoimaan CAN-väylän kautta päätietokoneelle.

Avainsanat: CAN-väylä, Qt, käyttöliittymä, Raspberry Pi, sähkömoottoripyörä

TABLE OF CONTENTS

1 INTRODUCTION

EBanditXamk is a school project that aimed for converting a conventional motorcycle into an electric-powered one. It was designed and developed by three electrical and automation engineering students. The project started at the end of the year 2019 and was completed In April of 2022.

The team for the project was selected by a competition where students were meant to draw up a preliminary plan for the motorcycle conversion. Our team which consisted of Me, Kalle Hellgren, and Miikael Riuttaskorpi won the competition.

The project started by searching for a motorcycle according to the preliminary plan. A suitable one was found in Helsinki for a reasonable price. This motorcycle had a faulty engine, which made it perfect for the project.

The objective of the project was to gain knowledge and data about electrical vehicles for Xamk. The secondary objective was to get teaching material for electronics and automation courses. Future students can study and develop the motorcycle even further.

1.1 Need for a self-made user interface

There were no displays available that would serve the needs of this project for sale. At the very beginning of the project, it came clear that one had to be designed for the motorcycle for it to meet the appearance and usability standards defined in the preliminary plan.

The minimal objectives set in the preliminary plan were that the driver should be able to see the most important information about the system on the display, such as speed, state of charge, and battery voltage. The physical installation should also be good looking for anyone who is not interested in electrical vehicles.

Many technologies could have been used for creating the user interface of Ebandit. This thesis describes how it was created and why the technologies used, were chosen to be used. This thesis is not intended to be a

programming tutorial but to be an informational writing about designing and developing an user interface for an electric vehicle. The source code can be found on Github.

1.2 EbanditXamk

The motorcycle chosen for this project was Suzuki Gsf 600, more familiarly known as Suzuki Bandit. Bandit was chosen because of its steel frame, which made the installation of the electric motor and the batteries easier. The second reason was that it is rather good-looking even without any modifications.

Image 1. Suzuki Bandit 600

Image 2. Ebandit at American Car Show

Images 1 and 2 show the difference between the original Bandit and the finished Ebandit. The use of outside work resources was as limited as possible. The only work that had to be done by people outside the team was the manufacturing of the battery enclosure, motor mounts, and the custom airbrush paint job of the battery enclosure.

Kalle Hellgren 3D-modeled the battery enclosure and the motor mountings which helped keeping the costs of the project lower. The work hours of the team members can be calculated in thousands of hours.

2 COMPONENTS AND TECHNOLOGIES

EbanditXamk has numerous electronic devices that need to be able to communicate with each other. These devices were selected after a thorough investigation of available products. One important selection criterion was that the devices could be configured to communicate with each other.

This chapter introduces the selected devices at a high level. Additional information about the electronics can be found in the manufacturer's official documentation.

2.1 Raspberry Pi

The heart of the system is Raspberry Pi 3 Model B+ with the Raspbian buster operating system.

Raspberry Pi is a single-board computer with an ARM Cortex-A53 1.4GHz processor and up to 1GB SRAM. It has built-in Wi-Fi and Bluetooth. With its four USB ports and HDMI-port, it is more than enough for this project. It has 40 GPIO-pins of most remain unused.

2.1.1 Raspbian

Raspbian is a free operating system based on Debian optimized for the Raspberry Pi hardware. An operating system is the set of basic programs and utilities that makes Raspberry Pi run /1/.

Raspbian was selected since it is basically a full Linux system. This allows the use of various Linux interfaces such as Socketcan which is needed to read and write CAN-frames with the Raspberry.

2.2 BMS

The battery pack of ElectricBandit is made from Lithium-ion batteries. These kind of batteries need BMS to operate safely. BMS monitors the condition of the battery pack and protects it in dangerous conditions such as low voltage or over-current.

The Orion BMS is a fully-featured lithium-ion battery management system that is specifically designed to meet the tough requirements of protecting and managing battery packs for electric vehicles (EV), plug-in hybrid (PHEV), and hybrid vehicles (HEV) with automotive-grade quality /2/.

2.3 Curtis motor controller

The motor controller provides power to the motor of the vehicle. It also measures the rotation speed of the motor with an encoder and the temperature of the motor with a thermistor.

The operation of the controller is controlled by BMS, thus for example, the voltage of the battery is getting too low, the BMS sends a command to the controller to lower its output power. Some parameters of the controller can be set from the Raspberry, such as Eco or Sport mode or the power of the regenerative braking.

The Curtis 'SE' AC controllers utilize the latest technology to increase the peak current ratings for each size of controller. For a given rating, the SE models are smaller and cost less than previous Curtis AC controllers, benefits that are highly advantageous for all types of applications /3/.

2.4 Elcon TC charger

The charger charges the battery pack when it is plugged in. It communicates with the CAN-bus. When the charger is plugged in the motor controller is programmed not to provide power to the motor.

2.5 Mcp2515 CAN shield

The Raspberry Pi needs an exterior CAN-shiel to interface with the CAN-bus of the motorcycle. This shield converts the CAN protocol to SPI which the processor of the Raspberry can read and write to.

There were industrial-grade CAN to USB repeaters available on the market but due to the shortage of components, this shield was selected for the project.

2.6 52PI display

The display used in this project was 5-inch capacitive touchscreen display manufactured by 52Pi. It was connected to the Raspberry Pi with HDMI and USB cables.

This display was selected to the project due to its high resolution and its size that made the installation of it easier.

2.7 Qt

The user interface was created using a tool called Qt. Qt is a cross-platform framework, which means that applications programmed with it can run on different devices regardless of operating systems. It is used by high-profile companies such as Mercedes-Benz and Koenigsegg which made it a perfect choice for this project.

The power of Qt comes from its ability to combine lower-level programming languages such as C++ with higher-level languages. In this project, all the business logic code is written with C++, and the visual parts are written with QML. This makes the application much faster compared to for example applications programmed with python.

Qt comes with a wide variety of libraries and APIs to serve the needs of a developer.

3 CAN-BUS

The communication between all the motorcycles devices was handled with a CAN-bus. Because CAN-bus is a great part of this project, it is important to cover it with more detailed information. The programming of the CAN-bus is covered in later sections.

3.1 History

CAN is a serial field bus protocol that was originally used in road vehicles. Its development history can be traced back to the early 1980s. At that time, all automotive manufacturers were using these point-to-point problems and wiring also systems enhance to the connect safety and electronic devices robustness, Bosch in vehicles. developed as the application of electronics increased rapidly, the wiring between different components became heavy, long, expensive, and disorganized; it also made repairs very difficult /4 p. 2/.

3.2 Can bus layers

CAN-bus can be divided into three layers which are the physical layer, the data link layer, and the higher layer. /4 p. 11/. This basically means wiring, bits running on the CAN-bus, and converting the bits to a human-readable form. Programming of the CAN protocol used in this project is introduced in chapter 5.2

3.2.1 Physical layer

According to the ISO 11898 standard, the CAN-bus consists of a twisted pair cable. One wire is CAN High which voltage range is 2.5 V to 4 V, the other is CAN Low which voltage is from 1 V to 2.5 V. The voltage difference represents values 0 (dominant) or 1 (recessive) /1 s. 12/.

The CAN-bus has always at least two nodes. The twisted pair cable must have a 120 Ω terminating resistor to minimize signal reflection /4 p. 13/.

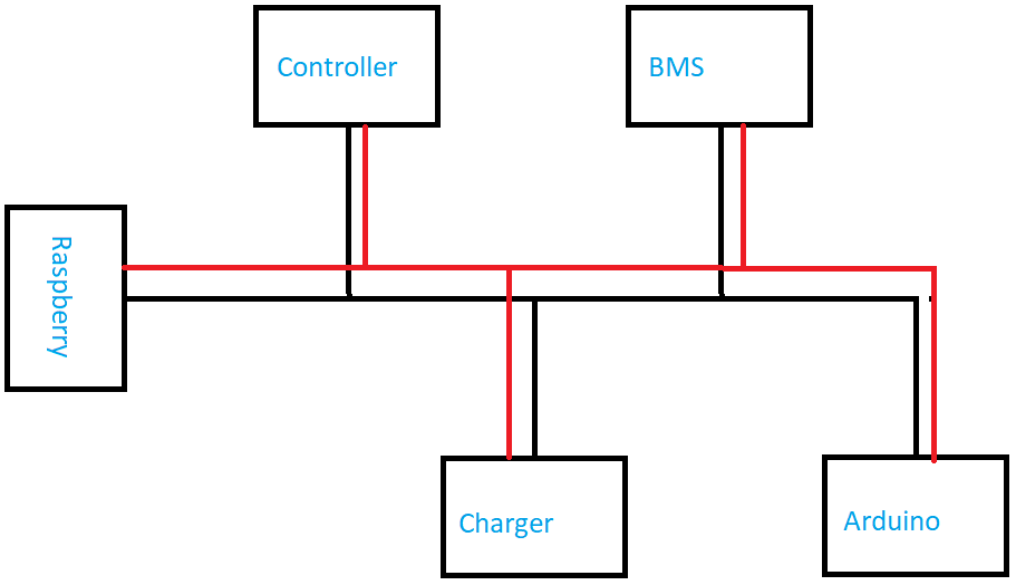

Image 3. Representation of the CAN-bus

Image 3 shows the planned CAN-bus wiring of the devices of the motorcycle. The terminating resistors were connected close to the BMS and the motor controller.

3.2.2 Data link layer

In the OSI reference model, the Data Link Layer (DLL) is one layer above the physical layer. Its function is to assemble the signals received from the physical layer into a meaningful message to provide a procedure for data transmission control /4 p. 15/.

3.2.3 Higher layer

The CAN Specification 2.0 and ISO 11898 jointly only specify the physical and data link layers in the OSI reference model. They do not define any tasks from the application layer, such as Start-up behavior, Identifier (ID) definition for different nodes, flow control, transportation of data exceeding data frame (eight bytes), data frame content definition, status reporting, and so on /4 s. 15/.

To supplement the CAN definition, there are associations, organizations, and companies specifying the higher layer protocol for CAN. The common protocols are CANopen (CAN in Automation), DeviceNet (Allen Bradley), CANaerospace, OSEK/VDX, etc. CAN Kingdom (KV ASER), J939 (by SAE), SDS, CANaerospace, OSEK/VDX etc /4 s.15/.

3.3 CAN frames

Messages that are sent via CAN-bus are called frames. A can frame sent from a device can be seen from other devices on the bus. Receiving devices will then decide if they are going to do any actions based on the frame or will they ignore it.

There are four types of CAN frames:

- Data frame
- Remote frame
- Error frame
- Overload frame

3.3.1 Data and remote frames

Contents of a data frame are:

- Start of the frame, which tells about a new frame.
- ID, Unique ID of the frame.

- Remote transmission request, 0 for sending data and 1 for requesting.
- Control field, data length.
- Data field, Actual data.
- CRC, Cyclic redundancy check, check that the frame is valid.
- End of the frame /5/.

Data and remote frame differ in that the remote frames data field is empty

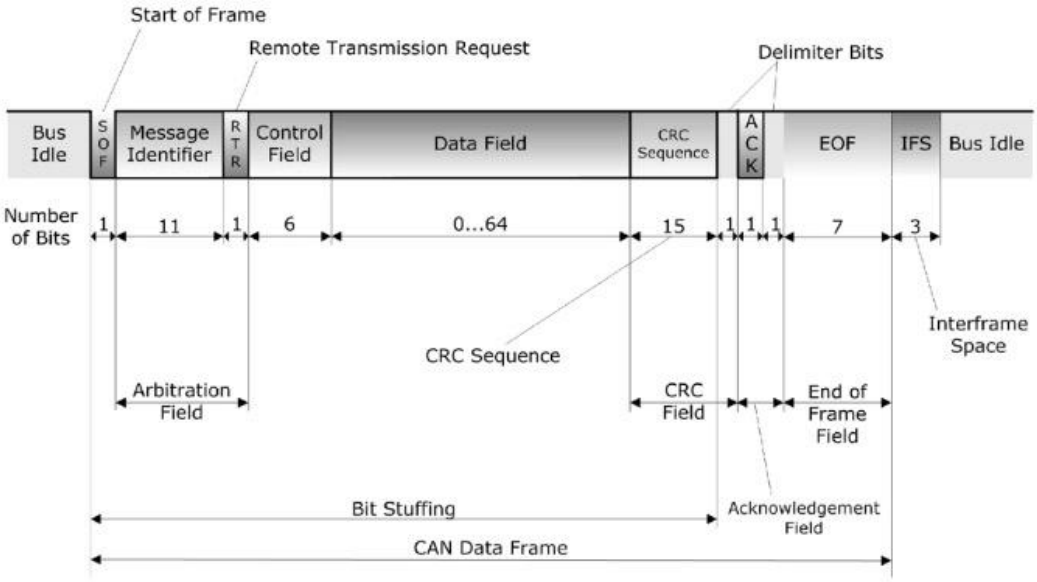

Image 4. Standard 11-bit ID can frame (Wilfried Voss 2018).

Image 4 shows the contents of a CAN frame. This would appear as zeroes and ones if viewed with an oscilloscope.

3.3.2 Error frame

The active error frame has six consecutive dominant bits. This frame is sent if a node detects an error condition on the CAN-bus. The other nodes on the bus detects this error frame and they will start transmitting error frames also /4 p. 21/. Image 3 below shows the contents of a error frame.

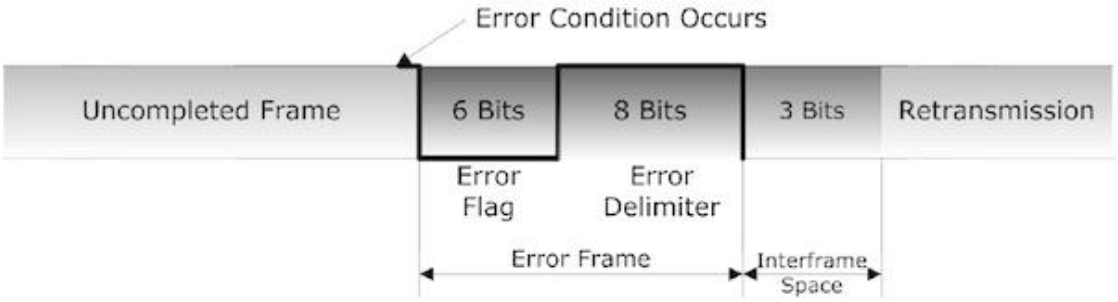

Image 3. Contents of error frame (Copperhill Technologies 2017)

3.3.3 Overload frame

The overload frame is used by the receiver to notify that it is not ready to receive any frames. Like an error frame, the overload frame is also comprised of two fields: an overload flag consisting of six to twelve dominant bits, and an overload delimiter consisting of eight recessive bits /4 p. 21/.

3.4 Prioritization

Devices can send data to the CAN-bus whenever it is available. If several devices are sending frames at the same time, the frame with the lowest ID will be sent and the others ignored.

4 DESIGN

This section covers the design process of the graphical user interface and the system architecture. The basic idea of this design was to make the system as modular as possible. This means that the development of the system can be done more easily in the future. One can simply design a device that sends its data via the CAN-bus and easily integrate it into the project.

4.1 Displayed values

The actual implementation part of this project started by gathering information about the devices running in the system. It is important to understand at least the very basics of the concepts behind all the devices sending information to the CAN-bus. This research process started before the motorcycle was bought and it took a good part of the design time of the whole project.

4.1.1 Battery values

The battery is the most crucial part of an electric vehicle. The driver needs to have information about the state of the battery during driving. The most important values that are shown on the screen are discussed in this chapter.

More information about the battery pack can be found in Kalle Hellgren's thesis or from at https://www.youtube.com/watch?v=UEFlmc8w0dg which is a video of manufacturing one smaller battery pack module for this project.

Lowest cell voltage

The battery pack of the motorcycle consists of 28 lion cells connected in parallel and 29 cells connected in series. This means that the voltage of one parallel-connected group is the same. The total voltage of the pack is the sum of the parallel-connected groups.

Lion cells can operate safely from around 2.7 volts to 4.2 volts. (These numbers can vary a little based on the battery type). The weakest link of the battery pack is the group that has the lowest voltage. This information is important for the driver since he/she can start driving economically when the voltage is getting lower. Of course, the battery management system is lastly in charge to cut the power when the voltage of one cell group is getting dangerously low.

Highest cell temperature

The battery pack was divided into five individual packs which were connected in series with copper busbars. One temperature sensor was installed in the middle of each pack. The highest temperature is displayed on the screen. The lithium-ion cell type that was used in this project can handle up to 80 °C of heat. As for the lowest cell voltage, the driver can act if the temperature is getting too high.

State of charge

State of Charge (SoC) describes how much the battery pack has capacity Ah or energy KWh left. It is calculated by the BMS with an algorithm that integrates the current of the battery pack over the charge or discharge cycle.

The capacity of the battery pack is 87Ah and the total energy is a little over 10kWh fully charged. SoC was displayed in ampere-hours in this case since it was more relevant than kilowatt-hours in this case. Another option would have been to display the SoC in percentage, but I preferred this option.

Current, voltage, power, and energy consumption

Current, voltage power, and energy consumption were values that wanted to be shown for the driver. The BMS sends these values to the CAN-bus at 20ms intervals, and they are updated immediately in the display.

The energy consumption [KWh/km] was calculated from the battery pack power and the time is taken for driving one kilometer using the formula:

$$U * I * \frac{1km}{v}$$

Where

U = voltage

I = current

v = is the velocity in km/h.

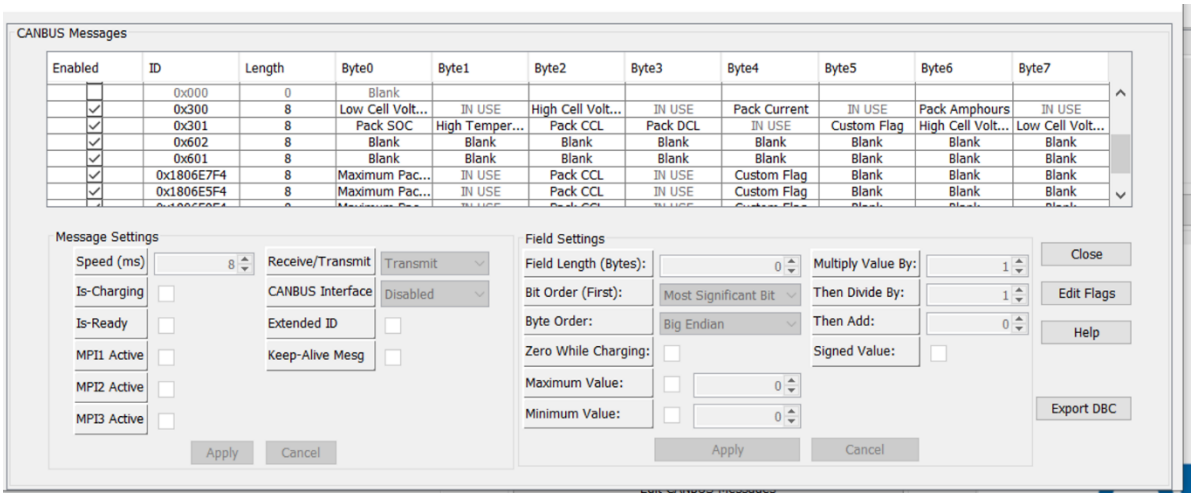

Image 4. BMS can frames

Image 4 illustrates the can frames that were configured to be sent from the BMS. Every value that the BMS is supervising can be sent to the CAN-bus.

4.1.2 Motor controller values

Like the BMS the motor controller is an autonomously functioning device that is programmed to give power to the motor and to communicate with other devices in the CAN-bus. This chapter describes only the values that are displayed on the screen. More information about the controller can be found in Miikael Riuttaskorpi's thesis.

Speed

The electrical motor that was used in this project has an integrated encoder that sends feedback to the controller. The controller converts this data to revolutions per minute and then sends it to the CAN-bus.

The rpm value can be converted to speed when the gear ratio of the motorcycle's sprockets and the perimeter of the rear wheel are known. The formula used for converting rpm to km/h is as follows:

$$\frac{\left(\frac{rpm}{Gr} * p * 60min\right)}{1000m}$$

Where

Gr = gear ratio

p = perimeter

This formula converts the rpm value to meters per hour. It is then divided by 1000 to get kilometers per hour.

Motor temperature

The motor has also an integrated temperature sensor whose data is also sent to the CAN-bus by the controller. The safe operating temperature of the motor goes up to 140°C

4.1.3 State of switches

All the motorcycle's physical switches were connected to an Arduino Uno, which controls the motorcycle's lights, hazards and horn. The Arduino is also programmed to send its data to the CAN-bus. The screen displays if the hazard is turned on the left or right and it also tells the driver which light is on.

This chapter describes the CAN-bus functionality from Arduino's perspective. More information about the source code and wiring of the Arduino can be found in Miikael Riuttaskorpi's thesis.

```
void sendCan() {
      tCAN message;

      message.id = 0x608;
      message.header.rtr = 0;
      message.header.length = 8;
      message.data[0] = headlightCanState;
      message.data[1] = blinkerCanState;
      message.data[2] = interlockCanState;
      message.data[3] = 0x0;
      message.data[4] = 0x0;
      message.data[5] = 0x0;
      message.data[6] = 0x0;
      message.data[7] = 0x0;

      mcp2515_bit_modify(CANCTRL, (1<<REQOP2)|(1<<REQOP1)|(1<<REQOP0), 0);
      mcp2515_send_message(&message);
}
```

Image 5. Arduino function for sending CAN frame

Image 5 illustrates a simple function for sending a CAN message based on the state of the switches of the motorcycle. The function works like this; it first initiates a variable which is type tCAN. (The type comes from the library that was used). Then an ID, remote transfer request, and data length fields are assigned to that variable. Note that the remote transfer request is 0 since no data is being requested.

Next data will be assigned for the data fields of the frame. The first three fields of the message had actual data while the others were only 0x0. For example, if the light switch is turned on low beam, then the headlightCanState variable would have the value of 0x01, at high beam, it would be 2 and 0 when the lights are turned off.

```
void readCan() {
 tCAN message;

 if (mcp2515_check_message())
 {
  if (mcp2515_get_message(&message))
  {
    if(message.id == 0x651)
    {
      ledCanState = message.data[0];
    }
   }
  }
}
```

Image 6. Arduino function for reading CAN frame

Image 6 shows a function for reading CAN-bus data with Arduino. This function checks if there are any messages in the CAN-bus. The working principle of this function is that it initiates a variable of type tCan. Then it checks if there are any frames in the CAN-bus and if there is, it calls for get_message function and passes a reference to that already created variable as its argument.

The get_message function uses SPI to convert the CAN-bus data into a format that the Arduino can use. It also assigns the needed fields for the message that can be then used in the Arduino code.

The next step is to check that the message has the ID that Arduino is interested in, in this case, it was 0x651. If a message with this ID was found its first data field would have been assigned to a variable that controls the led strips of the motorcycle.

```
// ----------------------------------------------------------------------------
uint8_t mcp2515_get_message(tCAN *message)
{
    // read status
    uint8_t status = mcp2515_read_status(SPI_RX_STATUS);
    uint8_t addr;
    uint8_t t;
    if (bit_is_set(status,6)) {
        // message in buffer 0
        addr = SPI_READ_RX;
    }
    else if (bit_is_set(status,7)) {
        // message in buffer 1
        addr = SPI_READ_RX | 0x04;
    }
    else {
        // Error: no message available
        return 0;
    }

    RESET(MCP2515_CS);
    spi_putc(addr);

    // read id
    message->id  = (uint16_t) spi_putc(0xff) << 3;
    message->id |=            spi_putc(0xff) >> 5;
```

Image 7 Part of the get_message function from the library

Image 7 is shown for illustrating the contents of a third-party library. This function is called from the Arduino's software described above.

4.2 Graphical layout

The user interface (UI) is the space where interactions between humans and machines occur. UI is an integral aspect of user experience (UX) that consists of two major parts: visual design, which conveys the look and feel of a product; and interaction design, which is the functional and logical organization of elements /6/.

UI design prioritizes the user's visual experience. A good user interface is functional, reliable, and enjoyable to use. User interface design should minimize the effort that the user has to invest interacting with a product and help users accomplish their goals with ease /6/.

The first goal was to create a layout that is easy to understand, and which describes all the necessary information about the devices in the CAN-bus.

This layout included gauges for speed and battery state of charge. Other values were presented with a representational icon that changes color based on

values retrieved from the CAN-bus. The actual values are displayed under the icons. Image 8 shows a demo version of the user interface running on a PC.

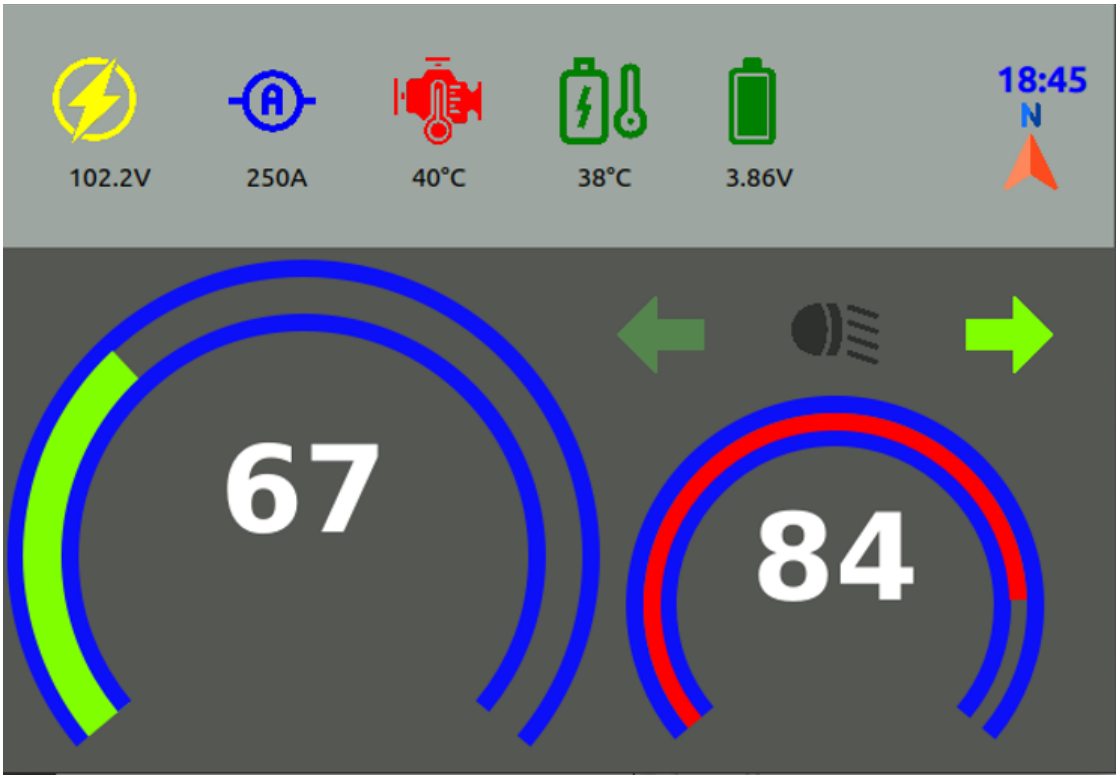

Image 8. The first version of the user interface

It is important to highlight the values with colors since the driver won't have too much time to inspect the display while driving. The values displayed from the top left are:

- Battery voltage.
- Battery current.
- Motor temperature.
- Highest battery temperature, there are a total of 5 thermistors.
- Clock and heading.
- Speed.
- Battery state of charge

The layout was programmed with QML language, and it was connected to the C++ backend using Qts methods.

4.3 Installation of the Screen

The screen needed to be installed in a such place where it could be easily read. The most natural place was to install it on the gasoline tank. Kalle Hellgren designed and 3D-printed a ring where the display could be installed

and I manufactured casing for it from a 1mm steel plate. The casing was MIG-welded onto the tank. The screen was then clued to the casing using Sikaflex for water insulation after the tank was painted.

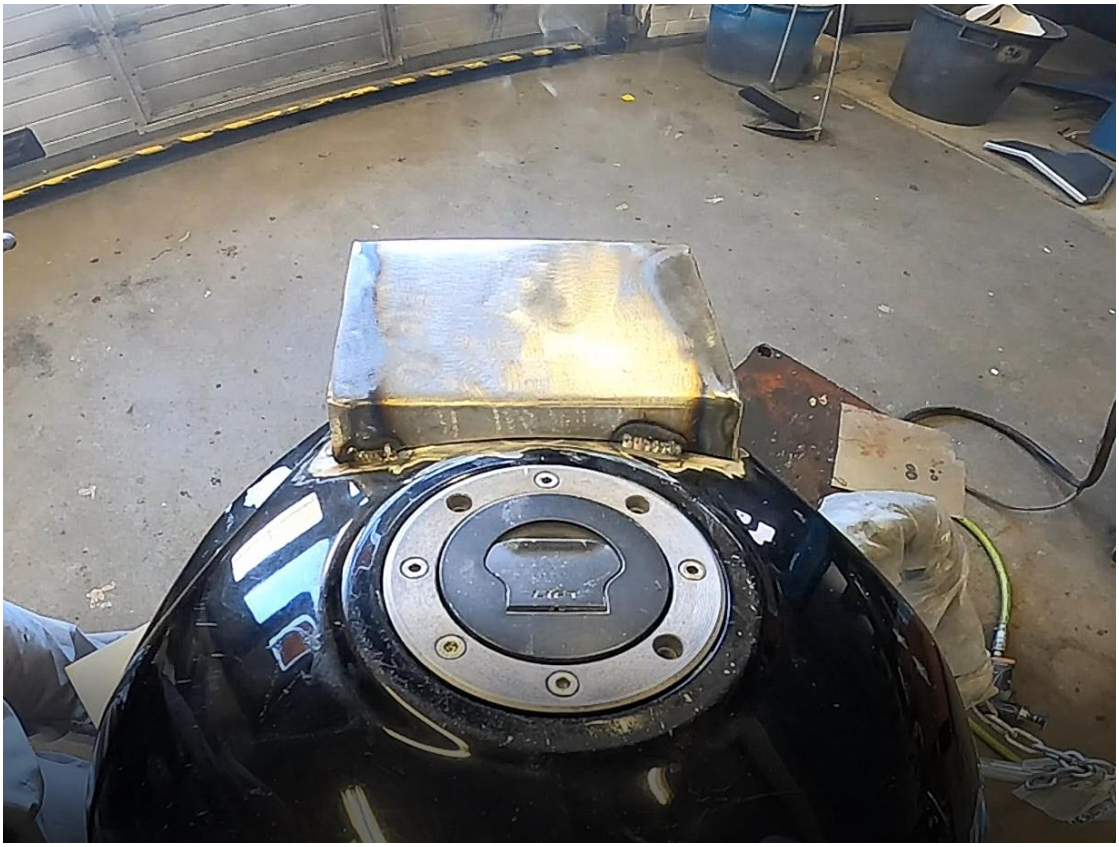

Image 9. Welding of the screen casing

An important note when welding thin metal is that the welding should be done in short tacks to prevent warping and burning through the metal. As figure 9 shows the casing was tacked for about 2 centimeters at a time and then switched to another position. This allows the weld to cool down.

Image 10. Painted tank

Image 10 shows the finished tank. The tank was cut hollow to make more space for the charger and junction box of the motorcycle. The metal underneath the screen was also cut to make it possible to plug or unplug the USB and HDMI cables of the screen.

The ring for the screen is visible in the top left corner of the image. More information about how the casing was made can be found in this video https://www.youtube.com/watch?v=b40mqFr8uNo.

5 QT LIBRARIES

The communication between the user interface and the devices on the CAN-bus was programmed using C++. As mentioned earlier Qt has many libraries and APIs to interface with the lower-level components and drivers of the target device. This chapter describes the classes and functions of the library. More information and programming examples are provided in chapter 6.4.

5.1.1 Qt Serial Bus

The Qt Serial Bus API provides classes and functions to access the various industrial serial buses and protocols, such as CAN, Modbus, and others. /7/.

5.1.2 QCanBus

QCanBus class was used to connect the application to the socketCan driver of the raspberry.

```
QString errorString;
QCanBusDevice *device = QCanBus::instance()->createDevice(
    QStringLiteral("socketcan"), QStringLiteral("vcan0"), &errorString);
if (!device)
    qDebug() << errorString;
else
    device->connectDevice();
```

Image 11. Example code provided by Qt (qt.io 2022)

5.1.3 QCanBusFrame

QCanBusFrame is a container class representing a single CAN frame. It contains the frame identifier and the data payload. QCanBusFrame contains the timestamp of the moment it was read /7/.

5.1.4 QCanBusDevice

QCanBusDevice communicates with a CAN plugin providing users with a convenient API. The CAN plugin must be specified during the object creation /5/.

This class contains the functions that are used for reading and sending CAN-frames. An object whose type is QcanBusDevice is created Programmatically and its functions are then used.

6 RASPBERRY PI PROGRAMMING

The programming of the screen started by installing the Raspbian operating system and configuring all the needed settings for the Raspberry. This chapter describes the necessary information for setting up the programming

environment and some programming examples. One is advised to find more information from other sources.

The knowledge needed to program the software was gained through the time undersigned was in school, from different courses, forums, technical documentation, and work-life. The learning curve was steep, but luckily Raspberry Pi and Qt have a great and open community where one can find help for their problems.

6.1 SSH connection

The programming of the raspberry was done via SSH connection using Linux PC. Raspberry Pi OS has the SSH server disabled by default. It can be enabled manually from the Raspberry's terminal using commands:

sudo raspi-config in a terminal window
Select Interfacing Options
Navigate to and select SSH
Choose Yes
Select Ok
Choose Finish /8/.

6.2 Connecting to internet

The Raspberry Pi must be connected to the internet for the SSH connection to work between the PC and the raspberry. The Raspberry can be connected to the internet by an ethernet cable or Wi-Fi. When using Wi-Fi, it is easiest to use the interface of the Raspberry to enter the Wi-Fi password.

Another option to connect to Wi-Fi is from the terminal of the Raspberry. Command "sudo nano /etc/wpa_supplicant/wpa_supplicant.conf" opens a configuration file where one can add Wi-fi credentials.

```
network={
    ssid="testing"
    psk="testingPassword"
}
```

Image 12. Example of network configuration. (Raspberrypi.com 2022)

Figure 12 represents an example network configuration after the config file was open. After saving and closing the file by pressing ctrl + x, the Raspberry should be booted, and it should be able to connect to the configured Wi-Fi network.

The next step was to find the Raspberry's IP address. It could be achieved in multiple ways, but the simplest was to type ''hostname -I'' in the Raspberry's terminal. This command returns the IP of the Raspberry, for example 192.168.1.8 and now it was possible to connect the PC by using command ''ssh pi@192.168.1.8", where pi is the name of the Raspberry, by default it is pi.

6.3 Software installation

Lots of software needed to be installed into the PC used in programming and into the Raspberry. This chapter describes the used software and their installation. The software used were mainly open source and the information about them can be found in official documentations of the software.

6.3.1 Qt

Qt was downloaded from the Qts official webpage https://www.qt.io/download and selecting the open-source version. This method downloaded Qt online installer. The installer was used to select and download the right version of the Qt which was 5.15.0 in this case. The installer also installed Qt creator IDE which was used to program the user interface.

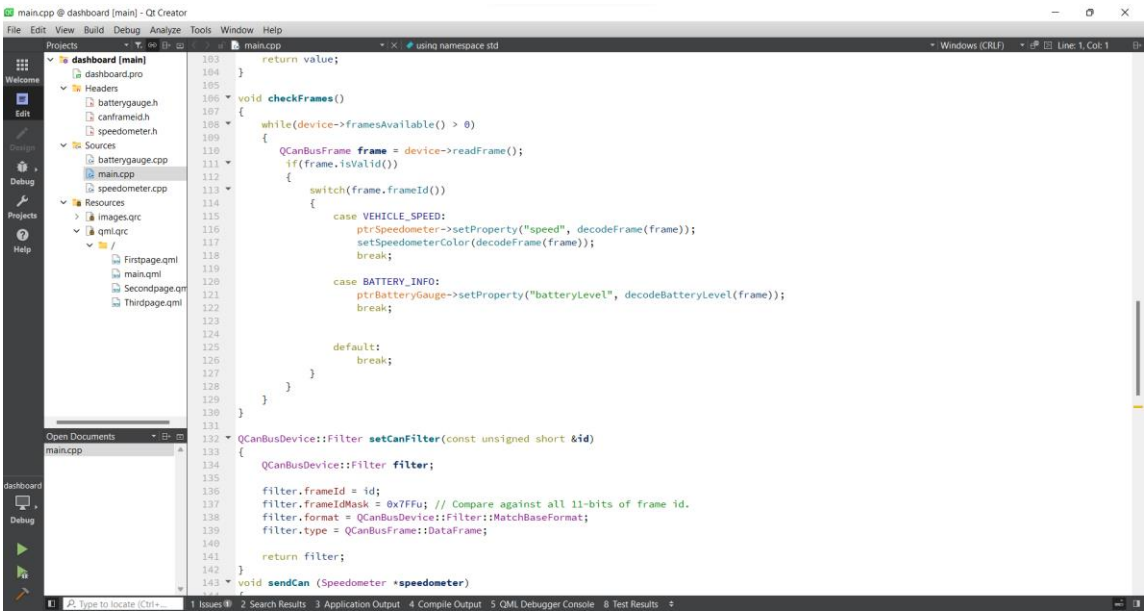

Image 13. Qt Creator IDE and some C++ code

Image 13 is a screenshot from the Qt creator illustrating the checkFrames function. This function is called every time that there are data on the CAN-bus. The function checks the validity of the frame first and then starts to do actions based on the frame's ID if the data was valid. This code was written at the early stages of the project, and it is different at the time of writing.

6.3.2 Compiler for raspberry

The system architecture of a Raspberry Pi is different than PCs which means that a cross-compiler needed to be installed to run the software on Raspberry Pi. The compiler and additional software packages were installed according to instructions found in this link https://github.com/UvinduW/Cross-Compiling-Qt-for-Raspberry-Pi-4.

6.3.3 Can-utils

Can-utils is a Linux-specific software package for interfacing the SocketCan driver of the Raspberry Pi kernel. The Raspberry needed some configuration to open the CAN communication.

The CAN-utils package was installed with "sudo apt-get install can-utils" command on Raspberry's terminal. After the can-utils was installed, it was needed to open /boot/config.txt file by typing "sudo nano /boot/config.txt" in the

terminal and adding “dtoverlay=mcp2515-can0,oscillator=8000000,interrupt=12”, “dtoverlay=spi-bcm2835-overlay” lines at the end of the file. Adding these lines in the file gives the MCP2515 integrated circuit in the CAN-shield instructions to function properly.

6.3.4 Bringing can interface up

The CAN-network was set up using command “sudo ip link set can0 up type can bitrate 250000” in the Raspberry’s terminal. This command sets the CAN network up at 250Kb/s speed. The speed was set to 250Kb/s because the motor controller uses that speed, and it cannot be configured to use any other speed.

A shell script was written to automate the process of bringing the CAN network up, and for starting the user interface software after the Raspberry was booted up correctly.

6.4 User interface programming

The easiest part of the project was the actual user interface programming. Most of the time was used for background work like choosing the right devices for the system or researching the documentation of the Qt framework. This chapter describes the principal ideas of the motorcycle’s UI software starting from the C++ side and ending on the QML side.

6.4.1 C++

The C++ side of the software handles all the CAN-bus communication, and it is also used to render the speed and SoC gauges of the screen. C++ assigns values to the QML front-end.

```
#include <QPainter>
#include "batterygauge.h"

Batterygauge::Batterygauge(QQuickItem *parent)
    :QQuickPaintedItem(parent),
      m_BatterygaugeSize(350),
      m_StartAngle(50),
      m_AlignAngle(260),
      m_LowestRange(0),
      m_HighestRange(100),
      m_Batterylevel(0),
      m_ArcWidth(15),
      m_OuterColor(QColor(12,16,247)),
      m_InnerColor(QColor(51,88,255,80)),
      m_BatteryLevelColor(QColor(255, 0, 0)),
      m_TextColor(QColor(255,255,255)),
      m_BackgroundColor(Qt::transparent)
{

}

```

Image 14. C++ class for displaying SoC

Image 15 shows a constructor for a C++ class that renders the state of charge of the battery. Variables that start with the m_ prefix are member variables or properties of the class.

```
qreal Batterygauge::getBatterygaugeSize()
{
    return m_BatterygaugeSize;
}

qreal Batterygauge::getStartAngle()
{
    return m_StartAngle;
}


qreal Batterygauge::getAlignAngle()
{
    return m_AlignAngle;
}


qreal Batterygauge::getLowestRange()
{
    return m_LowestRange;
}

qreal Batterygauge::getHighestRange()
{
    return m_HighestRange;
}
```

Image 16. Get function for the class properties

Get functions are used to get the value of the property. The get function is called when a property of this class is being accessed.

```
void Batterygauge::setBatteryLevel(qreal batteryLevel)
{
    if(m_Batterylevel == batteryLevel)
        return;

    m_Batterylevel = batteryLevel;
    update();
    emit batteryLevelChanged();
}
```

Image 17. Set function for property m_BatteryLevel

Set functions are called if one wants to change the value of the class property. The function checks that the value assigned to the property is different than it was already, and if it was different, it would assign the value and call for the update function that updates the value on the display.

Emit batteryLevelChanged is Qts special way to signal when an object's internal state has changed in some way that might be interesting to the object's client or owner /9/.

```
int main(int argc, char *argv[])
{
    QCoreApplication::setAttribute(Qt::AA_EnableHighDpiScaling);     'AA_EnableHigh

    QGuiApplication app(argc, argv);


    qmlRegisterType< Speedometer>("com.dashboard.speedometer",1,0,"Speedometer");
    qmlRegisterType< Batterygauge>("com.dashboard.batterygauge",1,0,"Batterygauge");

    QQmlApplicationEngine engine;

    engine.load(QUrl(QStringLiteral("qrc:/main.qml")));

    QObject *object = engine.rootObjects()[0];                              Don't
    qDebug() << object;

    QObject *speedometer = object->findChild<QObject*>("speedoMeter");
    QObject *batterygauge = object->findChild<QObject*>("batteryGauge");



    ptrBatteryGauge = qobject_cast<Batterygauge*>(batterygauge);
    ptrSpeedometer = qobject_cast<Speedometer*>(speedometer);
```

Image 18. Main function

The main function is where the software starts. All the needed objects are initiated in the main function. Also, the custom C++ classes are registered as a QML-types which makes it possible to render them on the display. Casting them as a QObject allows their values to be changed later in the code.

```
55  void handleFirstControllerInfo(const QCanBusFrame &frame)
56  {
57      const QByteArray payload = frame.payload();
58      const int highRpm = payload[0];
59      const int lowRpm = payload[1];
60      const int motorTempemperature = payload[2];
61
62      int motorRpm = (short)(((highRpm) & 0xFF) << 8 | (lowRpm));
63
64      speed = (motorRpm / 4) * 2 * 60 / 1000;
65
66      engineTemp->setProperty("text", QString("%1°")
67                              .arg(motorTempemperature));
68
69      ptrSpeedometer->setProperty("speed", speed);
70  }
71
```

Image 19. Function for setting displayed values

Image 18 shows a function that is called whenever there is a CAN-frame on the CAN-bus with an ID of 0x601. The function assigns the data field of the frame to a variable called payload which is a type of QByteArray. Next, it reads the values of the first three fields of the payload and assigns them to variables that are typed as integers.

The value for motors rotation is sent by two bytes and they had to be converted to a 16-bit value. The value of motor temperature was sent using one byte only.

The actual speed of the motorcycle is then calculated using the formula discussed in chapter 4.1.2. After converting the values to a readable form, they were assigned to the QML side of the software using the setProperty function.

```
4   //  CAN Bus IDs
5   #define FIRST_CONTROLLER_INFO  0x601
6   #define SECOND_CONTROLLER_INFO  0x602
7   #define FIRST_BMS_INFO  0x300
8   #define SECOND_BMS_INFO  0x301
9   #define THIRD_BMS_INFO  0x6B1
10  #define ARDUINO_INFO  0x608
```

Image 20. IDs of the CAN-frames of the system

The IDs of every CAN frame on the CAN-bus are described in image 19. The software has a function for every ID for decoding the bytes and displaying the values on the screen.

ID	Length	Byte0	Byte1	Byte2	Byte3	Byte4	Byte5	Byte6	Byte7	Count	Timestamp
253	8	01	10	68	01	C2	00	00	00	F9	20
254	8	40	1F	64	00	10	27	E8	03	FA	20
300	8	10	47	10	50	00	03	03	64	59	103
301	8	63	18	09	01	A4	00	06	00	59	104
601	8	00	00	19	1C	00	10	00	0D	46	108
602	8	00	00	11	00	03	DB	C2	00	46	107
608	8	01	00	2F	00	00	00	00	00	9E	14
651	8	04	00	00	00	00	00	00	00	66	99
6B1	8	7F	FC	2F	48	18	2F	48	3A	97	87

Image 21. Actual can data

Image 21 shows the actual CAN data of the system. The data was read with a CAN-bus reader. The equations for converting the data to a human readable form can be found in appendix 1.

6.4.2 QML

As already mentioned, the QML code is responsible for rendering the visual elements on the screen. The Qt creator has a drag and drop feature for creating user interfaces, but in this case, it was not used.

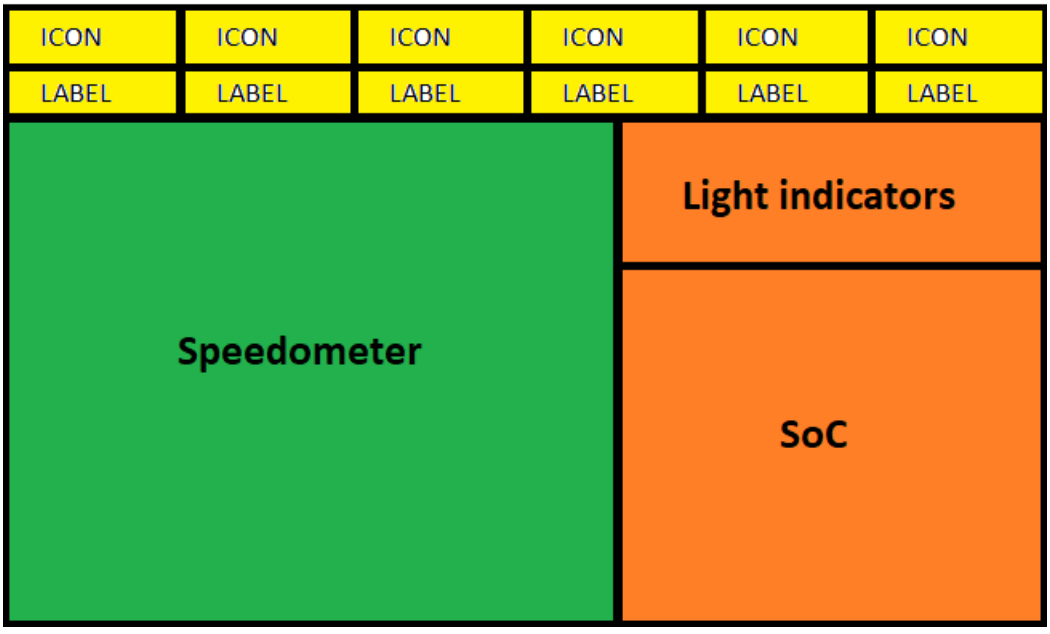

Image 22. The layout of the user interface

The items on the display are set on the screen as image 22 shows. The window was wrapped inside a column layout which makes the items appear on top of each other. The items in the top bar are wrapped inside a row layout that sets the items side by side. The speedometer, light indicators, and state of charge are wrapped in another row layout where light indicators and state of charge are in a column layout.

```
window {
    ColumnLayout {
        ToolBar{

        }

        RowLayout {
            Speedometer {

            }

            ColumnLayout {
                LightIndicators {

                }

                Batterygauge {

                }
            }
        }
    }
}

```

Image 23. QML code syntax

Image 23 shows the syntax of the QML language. The curly brackets represent the scope of the item. For example, all the items are inside the column layout but inside the row layout, there are only the items that are under the toolbar of the user interface.

```
Window {
    id: window
    visible: true
    width: 800
    height: 400
    color: "#555753"

    SwipeView {
        id: view
        currentIndex: 0
        anchors.fill: parent

        Firstpage {
            id: first
        }

        Secondpage {
            id: secondPage
        }
    }
}
```

Image 24. Main QML file

The main QML file is the frame of the user interface's QML software. As seen in image 24 the QML software is split into pages that display the information on the screen. The swipe view item makes it possible to use the touchscreen to change views of the user interface.

7 RESULTS AND CONCLUSIONS

After around 2000 hours of work from the team, it could be safe to say that the project was more than successful. The project gained lots of attention from conventional media and broke records of likes and views on Xamk's social media channels. Even Yle made a small TV news about the inspection of the motorcycle.

Testing of the motorcycle proved that the software of the user interface functions properly without bugs or crashes. It passed the inspection, and it is also easy to view while driving even in sunny weather when the sun is shining directly on the screen.

The user interface was developed simultaneously with other parts of the motorcycle which added some challenges to the programming of the user interface software. Also, Covid-19 delayed the project for about 6 months since the school was locked at that time. This caused that all the designed features could not be programmed onto the user interface within the time limits of the project.

Image 25. Finished user interface

Image 25 shows the user finished user interface. The values are clearly displayed to the user. The current and power are negative because the motorcycle is plugged in for charging.

If there would have been more time for developing the user interface, I would have developed a remote-control application for a mobile phone for visualizing

the motorcycle's data and for controlling it. This could have been done using MQTT communication between the Raspberry and the mobile phone.

The future of the motorcycle is to go around events related to vehicles or technology and to be a studying material for the future students of Xamk. I personally hope that one is going to develop the user interface more. If information is needed one can contact me for a briefing.

REFERENCES

[1] Raspbian documentation. WWW document. https://www.raspbian.org/ [referred 18.1.2022]

[2] Orion battery management system documentation. WWW document. Available at: http://www.orionbms.com/ [referred 18.1.2022]

[3] Curtis motor controller documentation. WWW document. Available at: https://www.curtisinstruments.com/products/motor-controllers [referred 19.1.2022]

[4] Zhu, Y. CAN and FPGA Communication Engineering: Implementation of a CAN Bus Based Measurement System on an FPGA Development Kit. Diplomica Verlag. 2010

[5] Wilfried, V. Controller Area Network (CAN Bus) - Message Frame Architecture. WWW document. Available at: https://copperhilltech.com/blog/controller-area-network-can-bus-message-frame-architecture/ [referred 18.1.2022]

[6] Adobe. UI design. Available at: https://xd.adobe.com/ideas/process/ui-design/ [referred 4.3.2022]

[7] Qt. Documentation of serial bus class. WWW document. Available at: https://doc.qt.io/qt-5/qtserialbus-index.html [referred 4.3.2022]

[8] Raspberry Pi documentation. WWW document. Available at: https://www.raspberrypi.com/documentation/computers/remote-access.html [referred 27.4.2022]

[9] Qt. Documentation for signals and slots. WWW document. Available at: https://doc.qt.io/qt-5/signalsandslots.html [referred 28.4.2022]

Appendix 1

Name	Short Name	Mode & PID	Equation	Minimum	Maximum	Unit	OBD2 Header	Scale Factor	Notes

Orion BMS Torque / EngineLink Extra PIDs/Sensor List

NOTE: Ewert Energy Systems is not affiliated with Torque or EngineLink in any way nor does it guarantee support for Torque or EngineLink. Torque and Enginelink are copyrighted by their respective owners. Please contact the authors of Torque or EngineLink directly for support regarding these software applications. Additionally, this list of PIDs is not intended to be an exhaustive list.

NOTE: The OBD header will need to be set for these PIDs to work properly in some vehicles / applications (this should be set to whatever the OBD2 ECU ID is set to). Also, the "ECM Simulation" field may need to be enabled in the battery profile depending on whether or not the vehicle has an Engine Control Module (ECM).

Last Updated: 7/27/2018

Torque Website: http://www.torque-bhp.com
EngineLink Website: http://www.ksolution.org/outdoor/enginelink.html

Long Name	Short Name	Mode & PID	Equation	Minimum	Maximum	Unit	OBD2 Header	Scale Factor	Notes
Charge Power Status	Charge Power	22F004	{B:7}	0	1	ON/OFF	See Note #1	x1	
Ready Power Status	Ready Power	22F004	{B:6}	0	1	ON/OFF	See Note #1	x1	
AM Power Status	AM Power	22F004	{B:5}	0	1	ON/OFF	See Note #1	x1	
Multi-Purpose Input	Multi-purpose Input	22F004	{B:4}	0	1	ON/OFF	See Note #1	x1	
Discharge Enable	Discharge Enable	22F004	{B:0}	0	1	ON/OFF	See Note #1	x1	
Charge Enable	Charge Enable	22F004	{B:1}	0	1	ON/OFF	See Note #1	x1	
Charger Safety	Charger Safety	22F004	{B:2}	0	1	ON/OFF	See Note #1	x1	
BMS Errors Present	Errors Present	22F004	{B:3}	0	1	ON/OFF	See Note #1	x1	
Balancing Active	Balancing	B206	A	0	1	ON/OFF	See Note #1	x1	
State of Charge	SOC	22F00F	A/2.0	0	100	%	See Note #1	x1	
Depth of Discharge	DOD	22F012	A/2.0	0	100	%	See Note #1	x1	
Pack Health	SOH	22F013	A	0	100	%	See Note #1	x1	
Pack Voltage	Pack Volt	22F00D	((A*256)+B)/10.0	0	350	V	See Note #1	x1	The maximum / minimums will vary based on how many cells are installed.
Pack Sum Voltage	Summed Volt	22F014	((A*256)+B)/100.0	0	350	V	See Note #1	x1	The maximum / minimums will vary based on how many cells are installed.
Pack Open Voltage	Open Volt	22F00E	((A*256)+B)/10.0	0	350	V	See Note #1	x1	The maximum / minimums will vary based on how many cells are installed.
Pack Resistance	Pack Res	22F011	((A*256)+B)/100.0	0	255	Ohm	See Note #1	x1	
Highest Cell Voltage	High Cell Volt	22F033	((A*256)+B)/10000.0	0	5	V	See Note #1	x1	Due to a Torque limitation, accuracy is reduced to 100mv.
Highest Cell ID	High Cell ID	22F03D	A	0	180	#	See Note #1	x1	
Lowest Cell Voltage	Low Cell Volt	22F032	((A*256)+B)/10000.0	0	5	V	See Note #1	x1	Due to a Torque limitation, accuracy is reduced to 100mv.
Lowest Cell ID	Low Cell ID	22F03E	A	0	180	#	See Note #1	x1	
Highest Open Cell Volt	High Open Cell V	22F036	((A*256)+B)/10000.0	0	5	V	See Note #1	x1	Due to a Torque limitation, accuracy is reduced to 100mv.
Lowest Open Cell Volt	Low Open Cell V	22F035	((A*256)+B)/10000.0	0	5	V	See Note #1	x1	Due to a Torque limitation, accuracy is reduced to 100mv.
Highest Resistance	High Res	22F039	((A*256)+B)/100.0	0	255	mOhm	See Note #1	x1	
Lowest Resistance	Low Res	22F038	((A*256)+B)/100.0	0	255	mOhm	See Note #1	x1	
12v Supply	Supply Volt	22F046	((A*256)+B)/10.0	0	20	V	See Note #1	x1	
Total Pack Cycles	Pack Cycles	22F018	((A*256)+B)	0	65535	#	See Note #1	x1	Total number of cycles put on pack
Battery Current	Amps	22F015	((((A*256)+B)-32767.0)/10.0)*-1	-500 (NOTE #2)	500 (NOTE #2)	A	See Note #1	x1	The maximum / minimums will change based on what current sensor is installed.
Battery Power	Battery kW		(VAL{Battery Current}*VAL{Pack Sum Voltage})/1000.0	-500	500	kW	BLANK	x1	
Charge Limit	CCL	22F00A	((A*256)+B)	0	500 (NOTE #2)	A	See Note #1	x1	The maximum / minimums will change based on what current sensor is installed.
Discharge Limit	DCL	22F00B	((A*256)+B)	0	500 (NOTE #2)	A	See Note #1	x1	The maximum / minimums will change based on what current sensor is installed.
Drive Mode (Prius)	Drive Mode	22F01F	A	0	6	#	See Note #1	x1	
Highest Temperature	High Temp	22F028	A	-40	80	C	See Note #1	x1	For Fahrenheit, change equation to: (9/5)*A+32
Lowest Temperature	Low Temp	22F029	A	-40	80	C	See Note #1	x1	For Fahrenheit, change equation to: (9/5)*A+32
Heatsink Temp	Internal Temp	22F0FF	A	-40	80	C	See Note #1	x1	For Fahrenheit, change equation to: (9/5)*B+32
Temperature #1	Temp 1	22F0FF	B	-40	80	C	See Note #1	x1	For Fahrenheit, change equation to: (9/5)*B+32
Temperature #2	Temp 2	22F0FF	C	-40	80	C	See Note #1	x1	For Fahrenheit, change equation to: (9/5)*C+32
Temperature #3	Temp 3	22F0FF	D	-40	80	C	See Note #1	x1	For Fahrenheit, change equation to: (9/5)*D+32
Temperature #4	Temp 4	22F0FF	E	-40	80	C	See Note #1	x1	For Fahrenheit, change equation to: (9/5)*E+32
Cell Voltage #1	Cell1	22F100	((A*256)+B)	0	5	V	See Note #1	x1	
Cell Voltage #2	Cell2	22F100	((C*256)+D)	0	5	V	See Note #1	x1	
Cell Voltage #3	Cell3	22F100	((E*256)+F)	0	5	V	See Note #1	x1	
Cell Voltage #4	Cell4	22F100	((G*256)+H)	0	5	V	See Note #1	x1	
Cell Voltage #5	Cell5	22F100	((I*256)+J)	0	5	V	See Note #1	x1	
Cell Voltage #6	Cell6	22F100	((K*256)+L)	0	5	V	See Note #1	x1	
Cell Voltage #7	Cell7	22F100	((M*256)+N)	0	5	V	See Note #1	x1	
Cell Voltage #8	Cell8	22F100	((O*256)+P)	0	5	V	See Note #1	x1	

Appendix 2/4

Name	Short Name	Mode & PID	Equation	Minimum	Maximum	Unit	OBD2 Header	Scale Factor	Notes
Cell Voltage #9	Cell9	22F100	((Q*256)+R)	0	5	V	See Node #1	x1	
Cell Voltage #10	Cell10	22F100	((S*256)+T)	0	5	V	See Node #1	x1	
Cell Voltage #11	Cell11	22F100	((U*256)+V)	0	5	V	See Node #1	x1	
Cell Voltage #12	Cell12	22F100	((W*256)+X)	0	5	V	See Node #1	x1	
Cell Voltage #13	Cell13	22F101	((A*256)+B)	0	5	V	See Node #1	x1	
Cell Voltage #14	Cell14	22F101	((C*256)+D)	0	5	V	See Node #1	x1	
Cell Voltage #15	Cell15	22F101	((E*256)+F)	0	5	V	See Node #1	x1	
Cell Voltage #16	Cell16	22F101	((G*256)+H)	0	5	V	See Node #1	x1	
Cell Voltage #17	Cell17	22F101	((I*256)+J)	0	5	V	See Node #1	x1	
Cell Voltage #18	Cell18	22F101	((K*256)+L)	0	5	V	See Node #1	x1	
Cell Voltage #19	Cell19	22F101	((M*256)+N)	0	5	V	See Node #1	x1	
Cell Voltage #20	Cell20	22F101	((O*256)+P)	0	5	V	See Node #1	x1	
Cell Voltage #21	Cell21	22F101	((Q*256)+R)	0	5	V	See Node #1	x1	
Cell Voltage #22	Cell22	22F101	((S*256)+T)	0	5	V	See Node #1	x1	
Cell Voltage #23	Cell23	22F101	((U*256)+V)	0	5	V	See Node #1	x1	
Cell Voltage #24	Cell24	22F101	((W*256)+X)	0	5	V	See Node #1	x1	
Cell Voltage #25	Cell25	22F102	((A*256)+B)	0	5	V	See Node #1	x1	
Cell Voltage #26	Cell26	22F102	((C*256)+D)	0	5	V	See Node #1	x1	
Cell Voltage #27	Cell27	22F102	((E*256)+F)	0	5	V	See Node #1	x1	
Cell Voltage #28	Cell28	22F102	((G*256)+H)	0	5	V	See Node #1	x1	
Cell Voltage #29	Cell29	22F102	((I*256)+J)	0	5	V	See Node #1	x1	
Cell Voltage #30	Cell30	22F102	((K*256)+L)	0	5	V	See Node #1	x1	
Cell Voltage #31	Cell31	22F102	((M*256)+N)	0	5	V	See Node #1	x1	
Cell Voltage #32	Cell32	22F102	((O*256)+P)	0	5	V	See Node #1	x1	
Cell Voltage #33	Cell33	22F102	((Q*256)+R)	0	5	V	See Node #1	x1	
Cell Voltage #34	Cell34	22F102	((S*256)+T)	0	5	V	See Node #1	x1	
Cell Voltage #35	Cell35	22F102	((U*256)+V)	0	5	V	See Node #1	x1	
Cell Voltage #36	Cell36	22F102	((W*256)+X)	0	5	V	See Node #1	x1	
Cell Voltage #37	Cell37	22F103	((A*256)+B)	0	5	V	See Node #1	x1	
Cell Voltage #38	Cell38	22F103	((C*256)+D)	0	5	V	See Node #1	x1	
Cell Voltage #39	Cell39	22F103	((E*256)+F)	0	5	V	See Node #1	x1	
Cell Voltage #40	Cell40	22F103	((G*256)+H)	0	5	V	See Node #1	x1	
Cell Voltage #41	Cell41	22F103	((I*256)+J)	0	5	V	See Node #1	x1	
Cell Voltage #42	Cell42	22F103	((K*256)+L)	0	5	V	See Node #1	x1	
Cell Voltage #43	Cell43	22F103	((M*256)+N)	0	5	V	See Node #1	x1	
Cell Voltage #44	Cell44	22F103	((O*256)+P)	0	5	V	See Node #1	x1	
Cell Voltage #45	Cell45	22F103	((Q*256)+R)	0	5	V	See Node #1	x1	
Cell Voltage #46	Cell46	22F103	((S*256)+T)	0	5	V	See Node #1	x1	
Cell Voltage #47	Cell47	22F103	((U*256)+V)	0	5	V	See Node #1	x1	
Cell Voltage #48	Cell48	22F103	((W*256)+X)	0	5	V	See Node #1	x1	
Cell Voltage #49	Cell49	22F104	((A*256)+B)	0	5	V	See Node #1	x1	
Cell Voltage #50	Cell50	22F104	((C*256)+D)	0	5	V	See Node #1	x1	
Cell Voltage #51	Cell51	22F104	((E*256)+F)	0	5	V	See Node #1	x1	
Cell Voltage #52	Cell52	22F104	((G*256)+H)	0	5	V	See Node #1	x1	
Cell Voltage #53	Cell53	22F104	((I*256)+J)	0	5	V	See Node #1	x1	
Cell Voltage #54	Cell54	22F104	((K*256)+L)	0	5	V	See Node #1	x1	
Cell Voltage #55	Cell55	22F104	((M*256)+N)	0	5	V	See Node #1	x1	
Cell Voltage #56	Cell56	22F104	((O*256)+P)	0	5	V	See Node #1	x1	
Cell Voltage #57	Cell57	22F104	((Q*256)+R)	0	5	V	See Node #1	x1	
Cell Voltage #58	Cell58	22F104	((S*256)+T)	0	5	V	See Node #1	x1	
Cell Voltage #59	Cell59	22F104	((U*256)+V)	0	5	V	See Node #1	x1	
Cell Voltage #60	Cell60	22F104	((W*256)+X)	0	5	V	See Node #1	x1	
Cell Voltage #61	Cell61	22F105	((A*256)+B)	0	5	V	See Node #1	x1	
Cell Voltage #62	Cell62	22F105	((C*256)+D)	0	5	V	See Node #1	x1	
Cell Voltage #63	Cell63	22F105	((E*256)+F)	0	5	V	See Node #1	x1	
Cell Voltage #64	Cell64	22F105	((G*256)+H)	0	5	V	See Node #1	x1	
Cell Voltage #65	Cell65	22F105	((I*256)+J)	0	5	V	See Node #1	x1	
Cell Voltage #66	Cell66	22F105	((K*256)+L)	0	5	V	See Node #1	x1	
Cell Voltage #67	Cell67	22F105	((M*256)+N)	0	5	V	See Node #1	x1	
Cell Voltage #68	Cell68	22F105	((O*256)+P)	0	5	V	See Node #1	x1	
Cell Voltage #69	Cell69	22F105	((Q*256)+R)	0	5	V	See Node #1	x1	
Cell Voltage #70	Cell70	22F105	((S*256)+T)	0	5	V	See Node #1	x1	
Cell Voltage #71	Cell71	22F105	((U*256)+V)	0	5	V	See Node #1	x1	
Cell Voltage #72	Cell72	22F105	((W*256)+X)	0	5	V	See Node #1	x1	

Appendix 3/4

Name	Short Name	Mode & PID	Equation	Minimum	Maximum	Unit	OBD2 Header	Scale Factor	Notes
Cell Voltage #73	Cell73	22F106	((A*256)+B)	0	5	V	See Note #1	x1	
Cell Voltage #74	Cell74	22F106	((C*256)+D)	0	5	V	See Note #1	x1	
Cell Voltage #75	Cell75	22F106	((E*256)+F)	0	5	V	See Note #1	x1	
Cell Voltage #76	Cell76	22F106	((G*256)+H)	0	5	V	See Note #1	x1	
Cell Voltage #77	Cell77	22F106	((I*256)+J)	0	5	V	See Note #1	x1	
Cell Voltage #78	Cell78	22F106	((K*256)+L)	0	5	V	See Note #1	x1	
Cell Voltage #79	Cell79	22F106	((M*256)+N)	0	5	V	See Note #1	x1	
Cell Voltage #80	Cell80	22F106	((O*256)+P)	0	5	V	See Note #1	x1	
Cell Voltage #81	Cell81	22F106	((Q*256)+R)	0	5	V	See Note #1	x1	
Cell Voltage #82	Cell82	22F106	((S*256)+T)	0	5	V	See Note #1	x1	
Cell Voltage #83	Cell83	22F106	((U*256)+V)	0	5	V	See Note #1	x1	
Cell Voltage #84	Cell84	22F106	((W*256)+X)	0	5	V	See Note #1	x1	
Cell Voltage #85	Cell85	22F107	((A*256)+B)	0	5	V	See Note #1	x1	
Cell Voltage #86	Cell86	22F107	((C*256)+D)	0	5	V	See Note #1	x1	
Cell Voltage #87	Cell87	22F107	((E*256)+F)	0	5	V	See Note #1	x1	
Cell Voltage #88	Cell88	22F107	((G*256)+H)	0	5	V	See Note #1	x1	
Cell Voltage #89	Cell89	22F107	((I*256)+J)	0	5	V	See Note #1	x1	
Cell Voltage #90	Cell90	22F107	((K*256)+L)	0	5	V	See Note #1	x1	
Cell Voltage #91	Cell91	22F107	((M*256)+N)	0	5	V	See Note #1	x1	
Cell Voltage #92	Cell92	22F107	((O*256)+P)	0	5	V	See Note #1	x1	
Cell Voltage #93	Cell93	22F107	((Q*256)+R)	0	5	V	See Note #1	x1	
Cell Voltage #94	Cell94	22F107	((S*256)+T)	0	5	V	See Note #1	x1	
Cell Voltage #95	Cell95	22F107	((U*256)+V)	0	5	V	See Note #1	x1	
Cell Voltage #96	Cell96	22F107	((W*256)+X)	0	5	V	See Note #1	x1	
Cell Voltage #97	Cell97	22F108	((A*256)+B)	0	5	V	See Note #1	x1	
Cell Voltage #98	Cell98	22F108	((C*256)+D)	0	5	V	See Note #1	x1	
Cell Voltage #99	Cell99	22F108	((E*256)+F)	0	5	V	See Note #1	x1	
Cell Voltage #100	Cell100	22F108	((G*256)+H)	0	5	V	See Note #1	x1	
Cell Voltage #101	Cell101	22F108	((I*256)+J)	0	5	V	See Note #1	x1	
Cell Voltage #102	Cell102	22F108	((K*256)+L)	0	5	V	See Note #1	x1	
Cell Voltage #103	Cell103	22F108	((M*256)+N)	0	5	V	See Note #1	x1	
Cell Voltage #104	Cell104	22F108	((O*256)+P)	0	5	V	See Note #1	x1	
Cell Voltage #105	Cell105	22F108	((Q*256)+R)	0	5	V	See Note #1	x1	
Cell Voltage #106	Cell106	22F108	((S*256)+T)	0	5	V	See Note #1	x1	
Cell Voltage #107	Cell107	22F108	((U*256)+V)	0	5	V	See Note #1	x1	
Cell Voltage #108	Cell108	22F108	((W*256)+X)	0	5	V	See Note #1	x1	
Cell Voltage #109	Cell109	22F109	((A*256)+B)	0	5	V	See Note #1	x1	
Cell Voltage #110	Cell110	22F109	((C*256)+D)	0	5	V	See Note #1	x1	
Cell Voltage #111	Cell111	22F109	((E*256)+F)	0	5	V	See Note #1	x1	
Cell Voltage #112	Cell112	22F109	((G*256)+H)	0	5	V	See Note #1	x1	
Cell Voltage #113	Cell113	22F109	((I*256)+J)	0	5	V	See Note #1	x1	
Cell Voltage #114	Cell114	22F109	((K*256)+L)	0	5	V	See Note #1	x1	
Cell Voltage #115	Cell115	22F109	((M*256)+N)	0	5	V	See Note #1	x1	
Cell Voltage #116	Cell116	22F109	((O*256)+P)	0	5	V	See Note #1	x1	
Cell Voltage #117	Cell117	22F109	((Q*256)+R)	0	5	V	See Note #1	x1	
Cell Voltage #118	Cell118	22F109	((S*256)+T)	0	5	V	See Note #1	x1	
Cell Voltage #119	Cell119	22F109	((U*256)+V)	0	5	V	See Note #1	x1	
Cell Voltage #120	Cell120	22F109	((W*256)+X)	0	5	V	See Note #1	x1	
Cell Voltage #121	Cell121	22F10A	((A*256)+B)	0	5	V	See Note #1	x1	
Cell Voltage #122	Cell122	22F10A	((C*256)+D)	0	5	V	See Note #1	x1	
Cell Voltage #123	Cell123	22F10A	((E*256)+F)	0	5	V	See Note #1	x1	
Cell Voltage #124	Cell124	22F10A	((G*256)+H)	0	5	V	See Note #1	x1	
Cell Voltage #125	Cell125	22F10A	((I*256)+J)	0	5	V	See Note #1	x1	
Cell Voltage #126	Cell126	22F10A	((K*256)+L)	0	5	V	See Note #1	x1	
Cell Voltage #127	Cell127	22F10A	((M*256)+N)	0	5	V	See Note #1	x1	
Cell Voltage #128	Cell128	22F10A	((O*256)+P)	0	5	V	See Note #1	x1	
Cell Voltage #129	Cell129	22F10A	((Q*256)+R)	0	5	V	See Note #1	x1	
Cell Voltage #130	Cell130	22F10A	((S*256)+T)	0	5	V	See Note #1	x1	
Cell Voltage #131	Cell131	22F10A	((U*256)+V)	0	5	V	See Note #1	x1	
Cell Voltage #132	Cell132	22F10A	((W*256)+X)	0	5	V	See Note #1	x1	
Cell Voltage #133	Cell133	22F10B	((A*256)+B)	0	5	V	See Note #1	x1	
Cell Voltage #134	Cell134	22F10B	((C*256)+D)	0	5	V	See Note #1	x1	
Cell Voltage #135	Cell135	22F10B	((E*256)+F)	0	5	V	See Note #1	x1	
Cell Voltage #136	Cell136	22F10B	((G*256)+H)	0	5	V	See Note #1	x1	

Appendix 4/4

Name	Short Name	Mode & PID	Equation	Minimum	Maximum	Unit	OBD2 Header	Scale Factor	Notes
Cell Voltage #137	Cell137	22F10B	((I*256)+J)	0	5	V	See Note #1	x1	
Cell Voltage #138	Cell138	22F10B	((K*256)+L)	0	5	V	See Note #1	x1	
Cell Voltage #139	Cell139	22F10B	((M*256)+N)	0	5	V	See Note #1	x1	
Cell Voltage #140	Cell140	22F10B	((O*256)+P)	0	5	V	See Note #1	x1	
Cell Voltage #141	Cell141	22F10B	((Q*256)+R)	0	5	V	See Note #1	x1	
Cell Voltage #142	Cell142	22F10B	((S*256)+T)	0	5	V	See Note #1	x1	
Cell Voltage #143	Cell143	22F10B	((U*256)+V)	0	5	V	See Note #1	x1	
Cell Voltage #144	Cell144	22F10B	((W*256)+X)	0	5	V	See Note #1	x1	
Cell Voltage #145	Cell145	22F10C	((A*256)+B)	0	5	V	See Note #1	x1	
Cell Voltage #146	Cell146	22F10C	((C*256)+D)	0	5	V	See Note #1	x1	
Cell Voltage #147	Cell147	22F10C	((E*256)+F)	0	5	V	See Note #1	x1	
Cell Voltage #148	Cell148	22F10C	((G*256)+H)	0	5	V	See Note #1	x1	
Cell Voltage #149	Cell149	22F10C	((I*256)+J)	0	5	V	See Note #1	x1	
Cell Voltage #150	Cell150	22F10C	((K*256)+L)	0	5	V	See Note #1	x1	
Cell Voltage #151	Cell151	22F10C	((M*256)+N)	0	5	V	See Note #1	x1	
Cell Voltage #152	Cell152	22F10C	((O*256)+P)	0	5	V	See Note #1	x1	
Cell Voltage #153	Cell153	22F10C	((Q*256)+R)	0	5	V	See Note #1	x1	
Cell Voltage #154	Cell154	22F10C	((S*256)+T)	0	5	V	See Note #1	x1	
Cell Voltage #155	Cell155	22F10C	((U*256)+V)	0	5	V	See Note #1	x1	
Cell Voltage #156	Cell156	22F10C	((W*256)+X)	0	5	V	See Note #1	x1	
Cell Voltage #157	Cell157	22F10D	((A*256)+B)	0	5	V	See Note #1	x1	
Cell Voltage #158	Cell158	22F10D	((C*256)+D)	0	5	V	See Note #1	x1	
Cell Voltage #159	Cell159	22F10D	((E*256)+F)	0	5	V	See Note #1	x1	
Cell Voltage #160	Cell160	22F10D	((G*256)+H)	0	5	V	See Note #1	x1	
Cell Voltage #161	Cell161	22F10D	((I*256)+J)	0	5	V	See Note #1	x1	
Cell Voltage #162	Cell162	22F10D	((K*256)+L)	0	5	V	See Note #1	x1	
Cell Voltage #163	Cell163	22F10D	((M*256)+N)	0	5	V	See Note #1	x1	
Cell Voltage #164	Cell164	22F10D	((O*256)+P)	0	5	V	See Note #1	x1	
Cell Voltage #165	Cell165	22F10D	((Q*256)+R)	0	5	V	See Note #1	x1	
Cell Voltage #166	Cell166	22F10D	((S*256)+T)	0	5	V	See Note #1	x1	
Cell Voltage #167	Cell167	22F10D	((U*256)+V)	0	5	V	See Note #1	x1	
Cell Voltage #168	Cell168	22F10D	((W*256)+X)	0	5	V	See Note #1	x1	
Cell Voltage #169	Cell169	22F10E	((A*256)+B)	0	5	V	See Note #1	x1	
Cell Voltage #170	Cell170	22F10E	((C*256)+D)	0	5	V	See Note #1	x1	
Cell Voltage #171	Cell171	22F10E	((E*256)+F)	0	5	V	See Note #1	x1	
Cell Voltage #172	Cell172	22F10E	((G*256)+H)	0	5	V	See Note #1	x1	
Cell Voltage #173	Cell173	22F10E	((I*256)+J)	0	5	V	See Note #1	x1	
Cell Voltage #174	Cell174	22F10E	((K*256)+L)	0	5	V	See Note #1	x1	
Cell Voltage #175	Cell175	22F10E	((M*256)+N)	0	5	V	See Note #1	x1	
Cell Voltage #176	Cell176	22F10E	((O*256)+P)	0	5	V	See Note #1	x1	
Cell Voltage #177	Cell177	22F10E	((Q*256)+R)	0	5	V	See Note #1	x1	
Cell Voltage #178	Cell178	22F10E	((S*256)+T)	0	5	V	See Note #1	x1	
Cell Voltage #179	Cell179	22F10E	((U*256)+V)	0	5	V	See Note #1	x1	
Cell Voltage #180	Cell180	22F10E	((W*256)+X)	0	5	V	See Note #1	x1	
HPEV RPM	HPEV RPM	B48601	((A*256)+B)	0	5000	RPM	See Note #1	x1	
HPEV Motor Temp	HPEV Motor Temp	B48601	C	-40	200	C	See Note #1	x1	
HPEV Control Temp	HPEV Cont Temp	B48601	D	-40	200	C	See Note #1	x1	
HPEV Amps	HPEV Amps	B48601	((E*256)+F)*10.0	-32767	32767	A	See Note #1	x1	
HPEV Voltage	HPEV Volts	B48601	((G*256)+H)	0	65535	V	See Note #1	x1	
HPEV Frequency	HPEV Frequency	B48602	((A*256)+B)	0	65535	Hz	See Note #1	x1	
HPEV Fault	HPEV Fault	B48602	C	0	255	Status	See Note #1	x1	
NOTE #1:	OBD2 Header will change based on what the Orion BMS OBD2 ECU ID is set in the profile. The default value when shipped is 0x7E3.								
NOTE #2:	Some of these values change based on the BMS configuration. For example, maximum and minimum current depend on the size of the current sensor used with the BMS.								