ELECTRIC VEHICLE CONVERSION PROJECT – HARDWARE IMPLEMENTATION AND DATA ACQUISITION SYSTEM

TEOH JIA XIAN

UNIVERSITI TUNKU ABDUL RAHMAN

ELECTRIC VEHICLE CONVERSION PROJECT – HARDWARE IMPLEMENTATION AND DATA ACQUISITION SYSTEM

TEOH JIA XIAN

A project report submitted in partial fulfilment of the requirements for the award of Bachelor of Engineering (Hons.) Electrical and Electronic Engineering

Faculty of Engineering and Science
Universiti Tunku Abdul Rahman

May 2016

DECLARATION

I hereby declare that this project report is based on my original work except for citations and quotations which have been duly acknowledged. I also declare that it has not been previously and concurrently submitted for any other degree or award at UTAR or other institutions.

Signature : _____

Name : TEOH JIA XIAN

ID No. : 12 UEB 02412

Date : 12 MAY 2016

APPROVAL FOR SUBMISSION

I certify that this project report entitled **"ELECTRIC VEHICLE CONVERSION PROJECT – HARDWARE IMPLEMENTATION AND DATA ACQUISITION SYSTEM"** was prepared by **TEOH JIA XIAN** has met the required standard for submission in partial fulfilment of the requirements for the award of Bachelor of Engineering (Hons.) Electrical and Electronic at Universiti Tunku Abdul Rahman.

Approved by,

Signature : _____

Supervisor : DR. CHEW KUEW WAI

Date : 12 MAY 2016

ELECTRIC VEHICLE CONVERSION PROJECT – HARDWARE IMPLEMENTATION AND DATA ACQUISITION SYSTEM

ABSTRACT

The main goal of the project focuses on the building the hardware for Electrical Vehicle Intelligent Monitoring System (EVICS). Developments of this projects include the usage of sensors, actuators, microcontrollers, microcomputers and electronic modules to support the hardware and software for the monitoring system. Coming from a vehicle originally fitted with internal combustion engine, it is realized that there exist several missing parameters that should be monitored with the fitting of the new electric motor and its controller. Furthermore, with the fuel running from batteries, it is evident that a proper vehicular information system needs to be established in order to provide the driver with the critical driving information, combining both electrical and conventional parameters, as the existing conventional gauges would not be much useful. This information can also be combined with a user interaction system and entertainment system to provide practical vehicle controls setup. The main interfacing systems that are involved in this project is the Raspberry Pi and CANBUS network. The CANBUS network has data fed from the electric vehicle controller, which provides essential information regarding the electric vehicle controls. Raspberry Pi is then used to collect data from various sensors and driver's feedback, including CANBUS data for further processing. The processed data is then output via a display to allow the driver to monitor their electric vehicle easily. Access to non-displayed parameters can also be done for mechanics and advanced users, which may review critical vehicular information which is logged systematically across time domain. A secondary system is also made available for driver or passenger use for non-critical components such as navigation and entertainment systems. As such, a comprehensive electric vehicle control system can be realized.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

LIST OF SYMBOLS / ABBREVIATIONS

A	ampere
AC	alternating current
ADC	analogue-to-digital converter
CAN	controller area network
CSV	comma-separated values
COB-ID	communication object identification
COD-ID	CAN object identifier
DC	direct current
EEC	European Union Regulation
EVICS	electric vehicle intelligent control system
GPS	global positioning system
GUI	graphical user interface
IoT	Internet of Things
ISO	International Organization for Standardization
LED	light-emitting diode
MCU	microcontrollers
NMT	network management transmission
PCB	printed circuit board
PDO	process data object
PWM	pulse width modulation
RPM	revolution per minute
SAE	Society of Automotive Engineers
SDO	service data object
SPI	serial peripheral interface
USB	universal serial bus
V	voltage

LIST OF APPENDICES

CHAPTER 1

INTRODUCTION

1.1 Background

With the increasing advancement of technology, the controls of a system become increasingly sophisticated, making the monitoring the system manually to be a rather meticulous process. It is, therefore, desirable to create an interface capable of overseeing and gaining control on the whole operation of the system, yet simplified enough where learning curve of system controls to be minimal for the average users. By using a user interface, the gaps between user operations and systems controls can be bridged, allowing users to empower the technology and the systems to unleash its full potential.

The leading of graphical user interfaces in commercial and industry products has proven that the consumers and market are in favour of such interface over traditionally based interfaces such as text or command-based interfaces. Today, graphical user interface is present in many of everyday digital devices that require interactions, from household devices such as television to home automation systems, communication devices such as wireless radios to smartphones, also catching up close are automotive systems. The days where drivers are only fed with driving information through gauges and warning lights are now replaced with digital gauges and display panels. Further integration with in-car entertainment systems also applauds the consumers, where drivers and even passengers are now able to feed in information to the systems as well, easily and swiftly via the graphical user interface. Such concept

can be summarized as an infotainment system. Vehicle settings and controls not only can be viewed but also altered on the fly, via usage of touch sensitive displays with system's haptic feedback as well. The same applies to the in-car entertainment systems where the roles of such system have increased to accommodate radio, optical players, GPS systems, video cameras, parking assistance and further integration to suit vehicle automated controls are even possible.

Besides viewing and controlling the vehicle's parameters, the complex integration of computer systems and hardware sensors network present also allows advanced vehicular monitoring systems, which keeps the vehicle conditions in the soundest way possible at all times. Commonly integrated sensors include manifold absolute pressure sensors for engine load sensing, wheel speed sensors for anti-lock braking systems, throttle position sensor for throttle input and tire pressure sensor for tire pressure monitoring. Abnormalities, warning, and hazards which are detected by the sensors will alert user via the graphical user interface or provide mechanical feedback; whereas for major or fatal issues, driver will be prompted for immediate input, and in real emergencies, the vehicular system may respond and take control based on sensors parameters, and return controls once the perceived hazard is over. It is no doubt that monitoring systems take the lead to play an essential role in active vehicle safety.

This project primarily focuses on the implementation of monitoring and control systems concepts into the Electric Vehicle Intelligent Control System (EVICS), which is currently developed electric conversion car project under the research of UTAR Centre of Vehicular Technology. Initially a conventional internal combustion engine vehicle, the dashboard gauges are therefore dated and catered specifically for the use of internal combustion engine, rendering most of the gauges and warning lights unused such as mechanical RPM meter, water coolant temperature, fuel gauge and check engine lights unusable. The provided display gauges from the electric vehicle controller are yet limited to a separate mechanical display for battery pack voltage, battery pack current, LED bar display on remaining level of the battery pack and a single line multi parameters display with a toggle switch to toggle between parameters. Such setup provides information to the driver in a rather clunky manner, not

mentioning reading of the gauges is difficult and clumsy during driving, and a backlight is virtually non-existent for the gauges during night driving, making the driving experience not pleasurable. A proposal to improve such situation includes a complete makeover of the existing vehicular information system, user interaction system and also entertainment system. Such setup would not only provide the driver with the vehicle information on the fly, yet at the same time be entertained by the music while being guided to the destination safely via control and navigation system, which ultimately makes the driving a pleasurable experience.

The conceptualization of the new EVICS systems includes the collection of vehicle data from vehicle controller's CANBUS system, which provides essential vehicle parameters, also utilizing the CANBUS system's CANopen communication protocol to read the electric vehicle controller's system parameters. The parameters, besides being viewed, are also being logged systematically, along with data from vehicular sensors network of the electric vehicle. Further controls of the vehicle are possible with the integration of actuators and sensors into the GUI, where GUI is not reserved only for the entertainment and navigation system, but also possible to allow simpler and autonomous operations of existing in-car systems. Safety measures of the electric vehicle systems are also integrated into the EVICS to ensure the driver is always notified for critical warnings for immediate corrective actions to be performed.

1.2 Aims and Objectives

The several proposals to improve the current monitoring systems include:

- To investigate vehicular parameters monitoring that is applicable to electric vehicle systems
- To design, construct and build the interfacing modules and install all the required components for vehicle systems monitoring and controls
- To analyse and improve existing electric vehicular system integration with external sensors network
- To obtain the complete vehicular controller's systems parameter via CANBUS
- To display the parameters of the obtained information and processed output from the vehicular systems

CHAPTER 2

LITERATURE REVIEW

2.1 Conceptualization of Electric Vehicle

Conceptualization of electric vehicle has come a long way with varieties of form factor. Originally based on rail vehicles, the idea has evolved into the marketable consumer vehicle segment and have spread to general commercial usage due to its known reliability and performance. Generally, electric vehicle consists of several main components, including the electrical energy source, electrical motor, and auxiliary components. The components are then linked up into through different interfacing via mechanical and electrical means to allow the system to work out harmoniously as an electrical vehicle.

Realizing the interfacing systems as the key role in linking up all the subsystems together, it is vital to ensure that each components parameters are able to be accessed, monitored and controlled individually to ensure the parameters are always in range of the optimal conditions. Furthermore, in this project which involves the conversion of internal combustion engine to a fully electric vehicle, the existing control systems present in the vehicle became cease in operation and incompatible with the new hardware setup. Therefore, a new EVICS system should be established to manage the communications of these systems and providing the user a proper interface for viewing and handling the vehicular systems parameters.

2.2 EVICS

Serving more than just a simple platform to replace the existing internal combustion engine controller in the electric vehicle, the EVICS also functions in acquiring, oversee and control all the duties to ensure the electric vehicle systems are in its best state at all times. The EVICS are expected to monitor the electric vehicle systems information such as battery monitoring systems, including battery information on battery voltage and current, cells health and charged state. Electric motor controls are also to be considered, where motor rotational speed, motor temperature, and motor current should be able to be deduced. Not to be left out is the conventional driving parameters such as speedometer, trip gauges, mileage gauges and visual or feedback warning system.

Similar concepts have been introduced in electric vehicle's manufacturers, where every particular detail has been paid to match the eccentric electric vehicle. Taking BMW i8 as an example, powered by powerful NVIDIA visual computing module, provides drivers with all the information required right on their dash as in Figure 2.1. From digital gauges to the entertainment systems to car handling systems, it's all in driver fingertips. The digital gauges constantly feeds drivers with the familiar information of the vehicle such as remaining travel range and next service appointments, while its entertainment system provides display with array of functions including an 8.8 inches touchscreen display, Bluetooth Audio Streaming and Handsfree Service, Smartphone Integration, navigation service, voice command option and real-time traffic information. BMW proprietary iDrive System also allows users to access vehicles computer to make changes directly to an on-board vehicular computer to suit their needs. Running on such platform requires massive computing power, which the NVIDIA module that provides stunning graphic and audio processing, capable of running a dual 4k display at once, at a much-reduced power consumption than a typical CPU.

Figure 2.1: BMW i8 dash display with virtual dials (Stevens, 2014)

2.3 Vehicular Parameters Sensing

2.3.1 Revolutions per Minute (RPM)

A tachometer is a dial of revolutions per minute served to show the engine crank rotational speed. Such readings allow drivers to select throttle level and gear settings to suit the dynamic driving conditions. A typical RPM gauge displays a range of values with some portion of the values placed in a red zone, where values in the red zone exceed the recommended safety limits of the engine speed.

Traditionally, the RPM is measured through a Hall Effect sensor placed near the engine coil leads from the low tension sides. Hall Effect is the measure of the generation of potential difference under a perpetually placed magnetic field perpendicularly across a conductor which carries electric current. The sensing component is usually placed near the high voltage ignition leads of the engine. By detecting the changes in deflection of electrons and magnetic flux intensity and further amplified with a processing circuitry, the RPM count can be calculated. This type of measurement provides an accurate, contactless yet relatively portable in size. Similar configurations can also be mounted on motor, gearbox, wheel or any rotating parts that are proportional to the motor speed, replacing the sensors with optoisolators instead

which uses light emission and photosensitive transducers, effectively eliminating sensing error from ferrous dust and are often reliable enough to include speed sensing as well.

Another approach in measuring the RPM of the engine is by determining the frequency on the alternator rotation. This alternator rotational output is also known as “AC Tap” tachometer output, which output is connected to the stator’s coil output before the rectifier. The signal output from this terminal as of Figure 2.2 consists of sine waves or sine waves with zero negative crossings are then processed as pulses. After dividing the pole of the alternator and evaluating the multiples of the frequency as compared to the alternator rotational frequency, the RPM reading can be obtained. However, the readings direct from the terminals can be quite filled with electrical surges and spikes, which may disrupt the reading values if not handled properly. Since the readings are based on alternator rotational input, it is important to ensure that the belts connecting the camshaft and the alternator always adhere to manufacturer's specification as any slippage or misalignment of belts may cause the alternator rotation speed to be affected, rendering the RPM readings to be flawed.

Figure 2.2: Alternator with AC tap output as indicated by the sticker label (Prestolite, n.d)

2.3.2 Motor Temperature

Motor temperature, one of the most vital parameters that keep the vehicle on the move, is rather sensitive to temperature rise. Standard terms used in the AC motor industry realized that the motor life span is halved for every 10°C rise of the motor's rated temperature. Therefore, to ensure the reliability of the motor, such parameter should be monitored close enough to ensure the temperature of the motor is always within its ceiling temperature.

However, it is also realized that measuring the temperature of the motor can also be difficult as its thermal distribution of the subsection of the motor depends on the motor's geometry and construction. It is noted that generation of heat from bearing frictional loss and electromagnetic loss should be the major account of temperature rise. Research also has shown that the highest temperature of the motor occurs is due to the low thermal conductivity of air and winding insulations, resulting the end windings of the motor to be the hottest region, which is denoted in Figure 2.3. (Kuria and Hwang, 2012) Such issue is not desired as the failing of windings will cause a major failure, causing issues to the electric vehicle users.

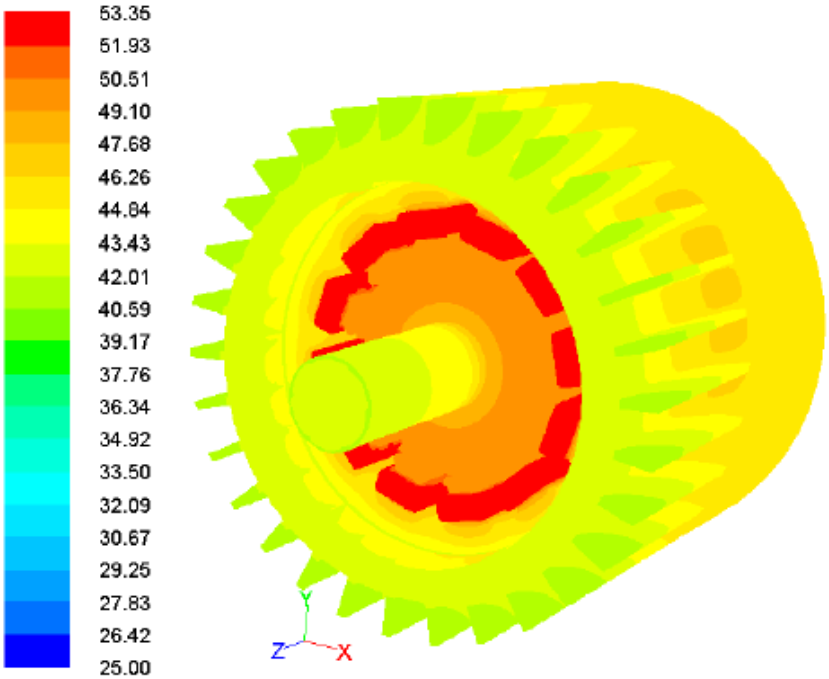


Figure 2.3: Contours of temperature of motor components (Kuria and Hwang, 2012)

An implementation of temperature sensors to the electric motor can be done to counter such issue. The temperature of the motor can be measured with thermocouples, thermistors and infrared measurement device, which offers either contact or non-contact measurement options. However, due to the construction of the electric motor where actual motor's temperature is determined by the hottest windings spot, checking the motor temperature via an external probe is rather inaccurate. Rather, the motor itself would provide a temperature sensor probes and its temperature signal will be sent for further processing. (Sarma and Nagaraju, 2012)

2.3.3 Controller Temperature

At the heart of the electric vehicle lies the motor controller which provides the main control systems on driving the electric vehicle, right from obtaining the power from the main battery pack, managing user controls, ramping up the voltage and current to drive the electric motor, not to mention also handling all the circuitry and settings that are involved within the electric vehicle systems.

Generally, a motor controller will be connected to a direct current (DC) battery source and a motor for drive systems. The motor controller will acquire power from the DC source and outputs a variable voltage or current supply needed by the electric motor to control the motor speed or torque. Such output also depends on a few parameters, including motor types, mapped motor response graph and user input. Modern electric vehicle controllers utilize advanced pulse width modulation techniques, making the electric drive system a high efficiency one as compared to older variable resistor arrangement. Also, can be expected from PWM drives include low motor harmonics and low torque ripple providing smooth speed control with minimized heating losses.

Such technology is realized by power transistor modules, allowing voltage and switching to be driven based on frequencies. However, one should realize that the high voltage or current transient, coupled with high motors inductance may cause quick and large heating effect on the power transistors, stressing the components. It is also

realized that once the controller temperature reaches a maximum tolerable value, the controller's performance will be throttled. To counter such issue, much designing and care have been taken during the design stage of the controller, yet the controller's heat issue is deemed unavoidable. Large heatsinks are usually installed in places with plenty of air cooling to ensure the performance of the motor is at its best. Further cooling can also be achieved by water cooling the motor with an external fan to draw the heat away from the motor at a certain temperature threshold.

Therefore, it is essential to ensure that the temperature of the motor controller is to be monitored closely. The mounting of such temperature sensors is critical to ensure the most accurate reading of the controller's temperature is obtained. For air cooled solutions, sensors mounting on the heatsinks or surfaces of the controller is possible. Such mounting will allow the readings to be acquiring the temperature of the controller surfaces accurately, allowing the temperature measurement to be focused evenly, and not directly from heat generation sources. For the water cooling, heat is usually transferred via liquid form to a heat exchanger, which then dissipates the heat via a natural or forced flow of air, and recirculated back. With the heat to be mostly transferred via the liquid coolant which has high thermal capacity, the temperature of the coolant can be used to determine the accurate temperature of the system. A temperature probe can then be mounted on the places where the coolant flows, and provides a reading of the liquid coolant temperature, also indirectly implying the temperature of the heating medium.

Also, there exists another way of obtaining the controller temperature data, which is via a semiconductor module. The module usually consists in integrated circuit packaging, allows heat to be radiated over them and provides high precision results, yet leaving an only small footprint in spaces, which is favoured in electronic circuit designs. This allows sensors to be mounted throughout an area without occupying much space, and can be directly mounted on a critical heat generating circuit components directly, unlike traditional thermistor sensors which are bulkier.

2.3.4 Controller Fault Monitoring

Vehicles on road today are mostly powered by vehicle computers, from engine controls, safety measurements to display instrumentations, the vehicle computer process and takes control of nearly every aspects of the vehicle. As such, the vehicle onboard computer can also potentially identify and deduce misleading data, which may allow further pinpoint to malfunction of elements or components of the vehicle.

During the appearance of a fault, the user display would normally provide an indicative light or warning to notify the driver that the vehicle has to be checked. Then, a diagnostic tool can be used to check the issue pertaining to the specific fault, which is normally indicated via codes, also known as fault codes. Then, the driver or mechanics can then take actions based on the fault codes provided, which normally indicates a specific form of malfunction or errors on specific devices or controls, minimizing the man-hours to troubleshoot and diagnose the vehicle systems.

Realizing the benefits, a fault monitoring system will be desired to allow instantaneous notification of fault and allow immediate diagnosis on the vehicle to be performed. The fault codes are usually generated by the vehicle's computer. The same rule applies to the electric vehicle controller, where the fault codes can be generated from ranging from own parameters such as faulty controller's sensors, overheating of the controller, to motor's parameters and battery supply parameters as well. The fault codes can then be checked against the manufacturer's list of fault codes to pinpoint the issues to be diagnosed.

2.3.5 Motor Current

Motor current are one of the key measurements in an electric vehicle as it relates the overall output and performance of the electric vehicle systems. To measure the current, several types of sensors can be deployed, which includes methods using a current-sensing transformer, Hall Effect current sensor, and a shunt resistor.

Current sensing transformer method utilizes a simple transformer which senses the measured current in the primary coil and energizes the secondary coil with a larger number of turn wounds. The number of turn ratios then determines the current output. The pros of such configuration are it can measure high current, yet at the same time providing galvanic insulation required. The disadvantage of this method is the transformer may saturate at high frequencies, limiting its used to only for low frequencies or constant frequencies applications. (Drafts, 2004)

On the other hand, shunt resistors methods are known to provide low cost yet accurate measurement. By using the method of calculating the voltage drop across a resistor of low value, the current can be calculated. However, such measurement is subjected to power dissipation in the resistance, which makes the method to be rather impractical for current more than 20 amperes.

Incorporating Hall generators, Hall Effect current sensors are easily integrated into embedded application. These solid state sensors are available in the form of integrated circuit packaging as in Figure 2.4, decreasing the space requirements of the sensing circuit and allow easy measurements to be carried out, where the sensor can be placed directly over the current trace on a printed circuit board (PCB). It is to be noted that the sensors itself are sensitive to temperature which may cause their accuracy to vary, but can be solved via closed-loop implementation. The closed-loop implementation of such sensor not only keeps the temperature drift low but also boast to offer far more superior accuracy, linearity, and wider frequency range. As such, implementation of the close-loop system may sound reasonable, however, one should also consider the costing to allow measurements in very wide current range. This is due to the implementation of more complex circuitry may be required to accommodate such requirements.

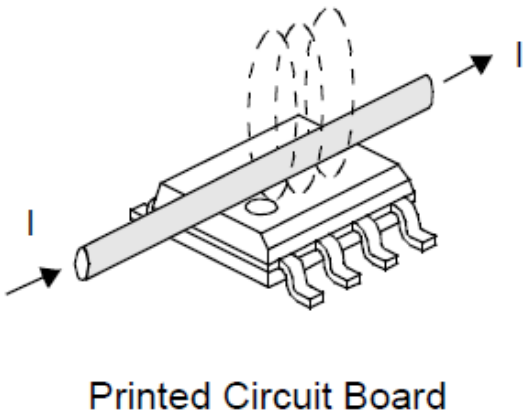


Figure 2.4: A PCB hall effect current sensor (Lepkowski, n.d)

2.3.6 Battery Voltage

It cannot be further emphasized the importance of monitoring battery conditions on a fully electric drive vehicle. The battery systems can be represented as fuel in a conventional internal combustion engine vehicle, where drivers heavily rely on such piece of information to ensure the vehicle is ready to go. For the electric vehicle control systems, the controller will interpret such information to optimize the best performance and range for the vehicle. The servicing technicians will also benefit from having an overall idea on the status of the battery system, which eases the maintenance and servicing of the electric vehicle.

One way of obtaining the overall battery voltage is via a construction of a voltage divider circuit and switched inputs to an amplifier, maintaining the voltage isolation between both high and low voltage circuits. The drawback, however, is the measurement errors arising from variations of transfer gains and non-linearity, which may occur due to the service life of components. Another issue is that the passing of analogue signals across optocoupler, additional amplifier output and analog-to-digital converter, the readings will suffer from measurement accuracy. (Manciac et al., n.d.)

Therefore, an improved method is proposed to measure such high voltage. An isolation barrier between high and low battery side is still maintained. The proposed

method as in Figure 2.5 involves the injection of periodic PWM signals via wave generator circuit, then the periodical signal is measured via digitization circuit. This type of measurement allows robust measurement even with the presence of disturbances in the high voltage system. (Magana and Veraguas, 2008) An analogue measurement circuit is still required to be constructed, and the sensing circuit is connected to a comparator input. Another input of the comparator is then connected to the PWM wave generator circuitry. The whole measurement is done on the comparator where the signals are fed in and compared, then produces an output which is a digital periodic signal, indicating the voltage of high voltage battery system. Such process allows the comparator output values to be transformed into a digital signal before passing through a digital isolation optocoupler, eliminating non-linearity and transfer gains variations. The measurement systems indicate a rather effective way to obtain the voltage value, however, the shortcomings reveal that the choice of an analogue periodic signal may affect the comparator output, therefore deviating the readings.

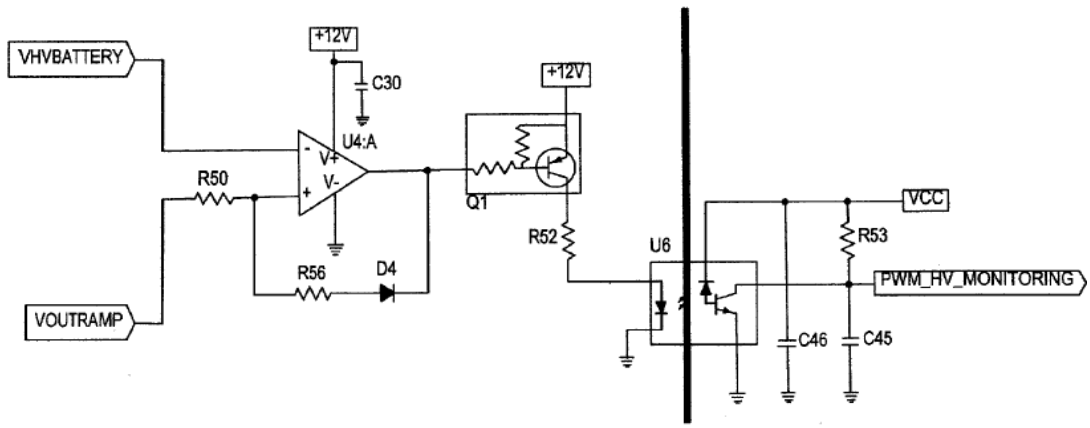


Figure 2.5: Measurement of battery voltage through PWM signal injection (Magana and Veraguas, 2008)

2.3.7 Throttle Input Sensing

Throttle input provides the variations of vehicle acceleration by the desire and control of the driver. Used to be a parameter based on mechanical linkages to link the throttle pedal and throttle valve, modern automotive vehicles now utilized throttle position sensors to ensure higher reliability and allowing more integrated functions for torque

management, such as stability control, traction control, cruise control and collision avoidance systems.

The construction of such input is based on inductive, Hall Effect sensors or magnetoresistive-based sensors. Such sensors can be mounted on the throttle pedal or at the throttle body. The sensor generally responds to the change of magnetic field by from throttle and the voltage generated from the sensing circuit, to be output and processed. Normally, a two pole rare earth magnet is used and considered to have a constant magnetic field that does not degrade from temperature and time.

Also present is the potentiometric type sensors. The principle of operation is similar to the use of a potentiometer, where a resistive strip is to be glided on by a metal brush as in Figure 2.6. The change in resistance will then be interpreted by the controller circuit and its output will be provided to the engine management system. It must be highlighted that safety concerns are still present as wear and tear and the presence of dirt may lead to inconsistent readings. Therefore, precautionary steps must be taken to ensure the safe operation of the vehicle in the event of failure of such sensors.

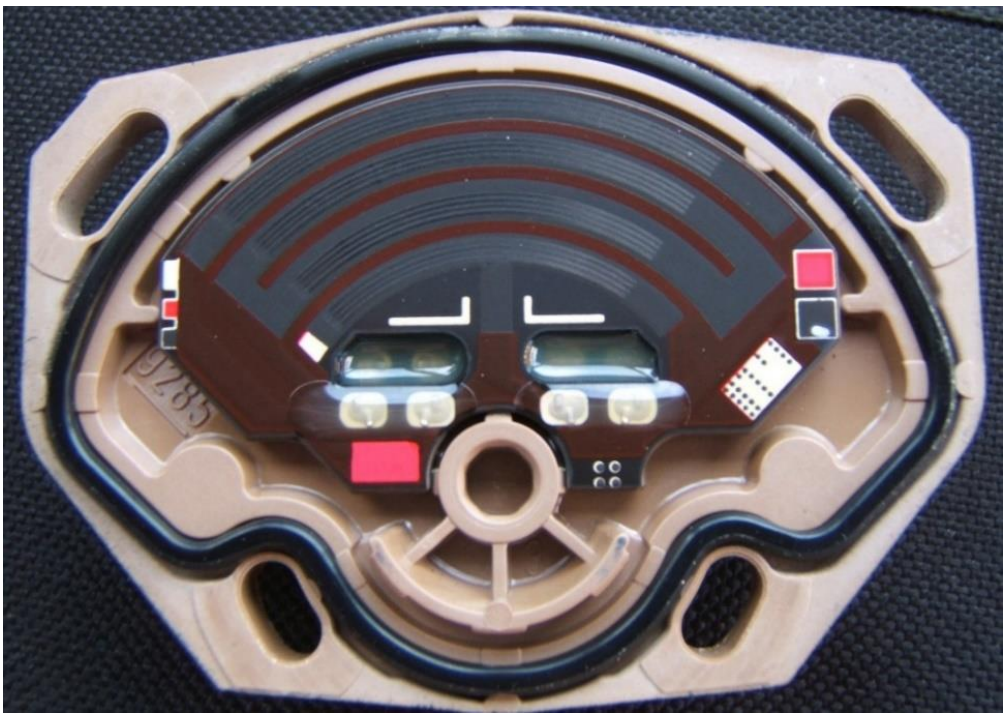


Figure 2.6: Internal view of a throttle position sensor, revealing resistive strips (Adrian, 2009)

2.3.8 GPS Location

GPS location can be determined by utilizing GPS modules that are readily available on the market. It is much favoured for its reliability and accuracy of determining location and time, not mentioning its free of charge. With standard communication via a serial protocol, the GPS modules will be able to suit different platforms provided as long as serial communication protocol is supported.

2.4 Vehicle Anti-Theft System

Posing as the basic theft-deterrent device, vehicle anti-theft systems today have evolved from the basic configuration with just emitting loud sound and strobing lights to having a starter kill switches, additional sensors trigger and even dual way paging controllers. Typical car alarm pinouts are indicated as in Figure 2.7.

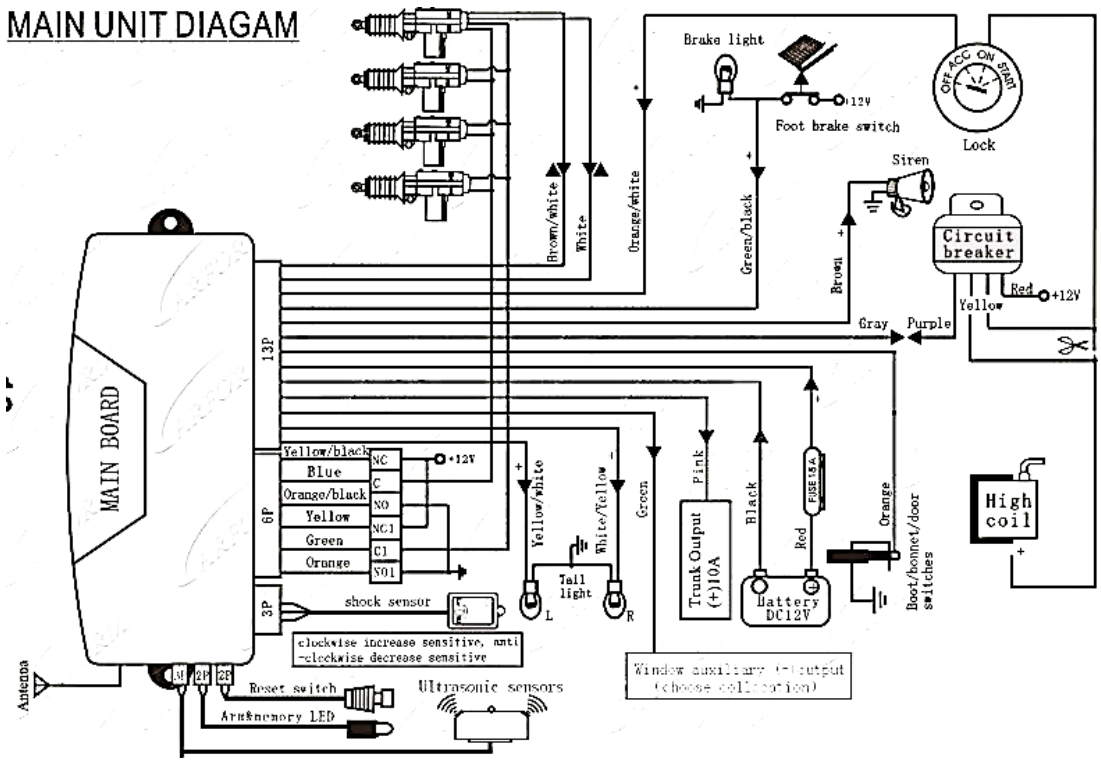


Figure 2.7: A typical car alarm pinouts (AlarmTek, 2012)

2.5 CANBUS

2.5.1 Overview

Controller area network, also known as CANBUS is a vehicle communication network standard that allows devices and microcontrollers to communicate with each other without having a central host computer. The bus works in a manner of broadcasting messages throughout the whole bus, and one can pick up the messages through any point of the system, also identified as a node of the system. Developed by Robert Bosch GmbH, the CANBUS protocol not only receives acknowledgements from Society of Automotive Engineers (SAE) but also quickly being adopted and finally becoming mandatory for all automotive vehicles in the United States and the European Union. The latter also receives improvements and restructuring by International Organization for Standardization (ISO) and further endorsed.

Specifically developed for automotive industry, the CANBUS systems bring all independent subsystems to communicate with each other through a channel, without truly having a component messages being more superior than other. In the communication bus, every component, or a node, get its turn to broadcast something into the channel, and all components are able to read all the data from the same channel as well, with the speed up to 1 Mbit/second. Figure 2.8 illustrates the arrangements. All the messages in the channel are also differentiated with priority bit. To differentiate one message from another, however, requires local filtering on CAN hardware on a specific node, allowing the component to analyse and may further react to the data. The system is also resilient when it comes to error handling, where error counters are available to all the nodes, which is all of the components in the system. Therefore, any problems discovered will raise the error flag by the node and the channel will react as accordingly to discard the error messages and attempt to send or retrieve the data again. An error counter is also available, allowing the sender and receiver to track the data error frequency. Once the data error frequency exceeds a predefined threshold, the transmitting node will become error passive and then goes fully off the channel to allow other bus traffic to flow uninterrupted.

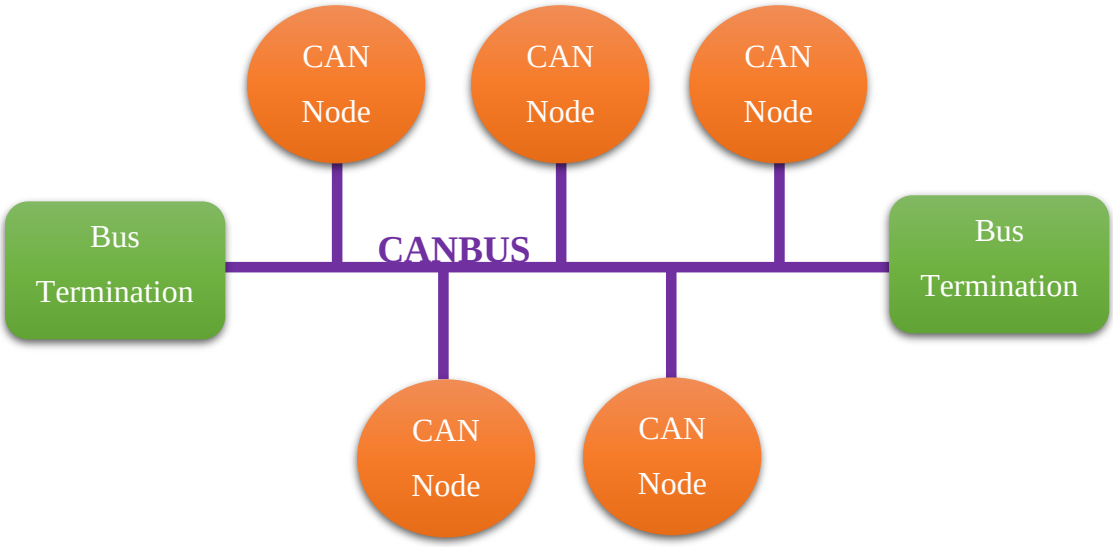


Figure 2.8: CANBUS Interfacing and Its Components

2.5.2 Higher Layer Protocols

The CAN protocol standard has provided a guideline on the method to transfer data from a point to another using a shared communication medium. However, a high layer protocol is also desired to manage the intersystem communication, to name a few, flow control, transportation of data, the establishment of communication and node address specifications. Generally, the higher layer protocol defines the behaviour upon start-up, message identifiers distribution on different nodes in the systems, data frames content translation and the status reporting within the system. The standards vary over the protocols and may be customized and extended by different manufacturers, ranging from automotive industry which includes CAN Kingdom, CANopen, GMLAN, SAE J1939 to marine and aviation CANaerospace, NMEA 2000 and even building automation VCSP.

2.5.3 CANopen

The CANopen protocol has provided a standardized communication between different nodes of devices on a network, even across different manufacturers. With the physical

hardware and data link protocol already established by the CAN standards, as shown in Figure 2.9, CANopen umbrella covers the networking, transportation of data, data sessions, data representation and finally allow application of the data.

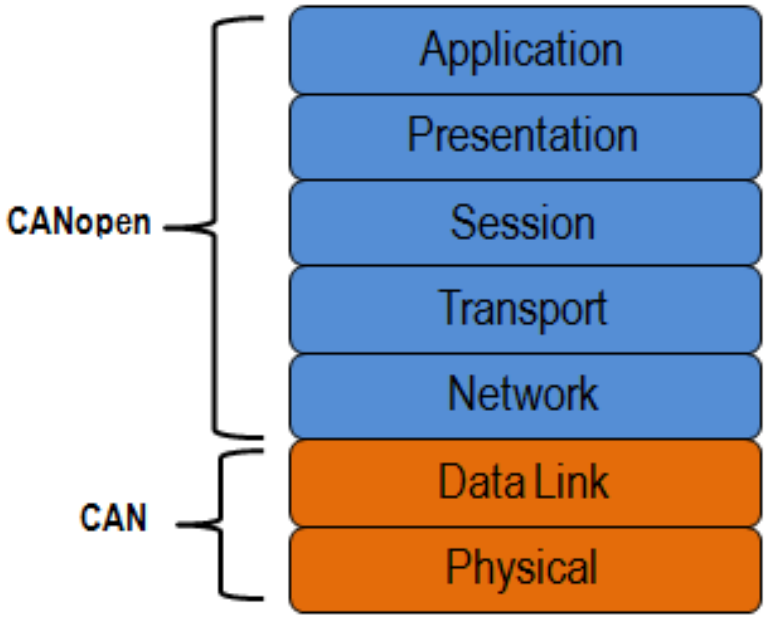


Figure 2.9: Representation on Coverage of Systems in CAN and CANopen

A CANopen compliant device should also conform to the standards of operation of its software. (NI, 2013) A communication unit is first required to allow interfacing of messages with various nodes in the network through the desired protocol. The minimum state machine is also required where all the initialization and resets of the device are well controlled and cycled through pre-defined configurations. Besides, the device should also hold an object dictionary which can utilize the data from the network or reflect its own data. The data consists of a 16-bit index of an array variable and expands to additional 8-bit sub-index for each variable. Finally, the device will reach application mode once the device is in an operational state, where object dictionary's variable will be modified and further receiving and transmission of data along the network are performed.

For a device to communicate with the CANopen network, the device must first have the same baud rate with the network to allow successful communications. Its

node ID shall also be assigned for identification, which allows other devices to identify and communicate with the device, and also, allow differentiation of messages sent and received amongst other bus devices. The node ID also sets its priority in bus messages and bus arbitration and is part of a requirement by CAN Object Identifier (COD-ID). As in Figure 2.10, the standard message types in the COD-ID are defined to 4 upper bits out of 11 bits, making 16 types of messages possible. The bottom 7 bits are filled with Node ID of the devices. An example of a standard message type is also displayed in Table 2.1.

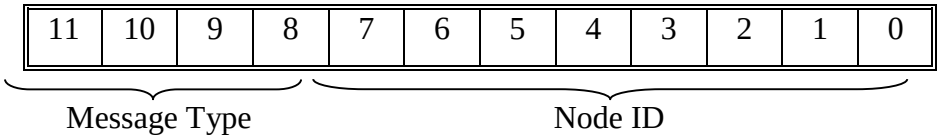


Figure 2.10: CANopen-Compliant COB-ID Message Organization

Table 2.1: Standard Message Type for Curtis Controllers (Curtis, 2014)

Generic Type	Message Identifier	Value (binary – hex)
NMT	NMT	0000 – 0x0
EMERGENCY	SYNC_ERR	0001 – 0x1
PDO	PDO1_MISO	0011 – 0x3
	PDO1_MOSI	0100 – 0x4
	PDO2_MISO	0101 – 0x5
	PDO2_MOSI	0110 – 0x6
SDO	SDO-MISO	1011 – 0xB
	SDO_MOSI	1100 – 0xC
HEARTBEAT	NODE	1110 – 0xE

To acquire messages from the CANopen, one can utilize the Service Data Object (SDO) protocol. Messages transmitted may include Network Management Transmission (NMT), which are of highest priority. This allows controls on the node's device state, detect boot-up and detect error conditions within the network. The second

highest priority in the protocol of CANopen is the emergency messages. The messages content includes the hour meter, specific fault, error category and error register for diagnosis. Heartbeat messages are also sent periodically by each node on the bus to indicate the device status and are of lowest priority. Process Data Object (PDO) protocol are also listed as one of the message types. This messages have medium priority and can transfer 8 bytes in a packet of data into or out of the device. This 64-bit data can carry messages of the various nodes such as output command bytes, status bytes, digital inputs and analogue inputs bytes in a single packet of data, which conserves bus bandwidth. Service Data Object (SDO) is another protocol defined by CANopen to view and alter the parameter data. They are usually of low priority and are infrequently used. This protocol is normally utilized only to retrieve rather infrequently used basic information such as manufacture dates and manufacturing system revisions. Fault log reviewing and key internal variables monitoring which are only catered exclusively for system debug purposes are also available in this protocol. In the Curtis electric vehicle controller, the service protocol for SDO can accommodate more than just reading data out of the protocol, it can also be utilized to set system parameters such as operation mode, current limiting, PWM limiting, gain factor and more. Table 2.2 explicitly lists some of the available parameters accessible. This allows the SDO to be used more than just diagnostic purposes, but also able to control some of the internal vehicle parameters. However, it is realized that retrieving data from SDO can be meticulous in terms of setting up for requesting data and deciphering received data, not mentioning writing the data into the protocol itself.

Table 2.2: Part of Standard Message Type for Curtis Controllers, Accessible via SDO (Curtis, 2014)

Parameter Profile Object Dictionary				
PARAMETER	SDO LOCATION INDEX	SUB-INDEX	RANGE *CAN VALUE*	DESCRIPTION
Operation Mode	0x3000	0x01–0x09	0–3, 5–7 *0–3, 5–7*	Driver mode: 0 = Active High Digital Input mode. 1 = Constant Current mode. 2 = Constant Voltage mode. 3 = Direct PWM mode. 5 = Constant Current mode, with open detection. 6 = Constant Voltage mode, with open detection. 7 = Direct PWM mode, with open detection.
Current Limit	0x3001	0x01–0x09	0.00–3.00 A *0–300*	Sets the maximum current output when the PDO command is 100% (255), **when operating in Constant Current mode**.
PWM Limit	0x3002	0x01–0x09	0–100.0 % *0–1000*	Sets the maximum PWM output when the PDO command is 100% (255), **when operating in Direct PWM mode**.
Voltage Limit	0x3003	0x01–0x09	0.0–36.0 V *0–360* (36V models) 0.0–80.0 V *0–800* (80V models)	Sets the maximum voltage output when the PDO command is 100% (255), **when operating in Constant Voltage mode**.
Dither Period	0x3004	0x01–0x09	4–200 ms *4–200*	Sets the time between dither pulses for each output (in 2ms steps). A dither period of 4–200ms provides a frequency range of 250–5Hz. Applicable only in Constant Current mode.
Dither Amount	0x3005	0x01–0x09	0–500 mA *0–500*	Sets the amount (+/-) of dither that will be added/subtracted from the command (in 10mA steps). Applicable only in Constant Current mode.
Kp	0x3006	0x01–0x09	0.1–100.0 % *1–1000*	Sets the proportional gain factor of the PI current controller.
Ki	0x3007	0x01–0x09	0.1–100.0 % *1–1000*	Sets the integral gain factor of the PI current controller.
Nominal Voltage	0x3010	0x00	12.0V–36.0V *120–360* 36.0V–80.0V *360–800*	Sets the nominal battery voltage, which is used in fault detection. 1353-4101/4001: 12V, 24V, 36V. 1353-6101/6001: 36V, 48V, 60V, 72V, 80V.
Analog Input Type	0x3020	0x00	0–63 *0–63*	Sets the input type on Analog 1 through 6. LSB is for Analog 1 and next is for Analog 2, etc. Upper two bits are not used. Bit = 0, voltage input type. Bit = 1, resistive input type.
High Threshold	0x3021	0x01–0x06	0.0–15.0 V *0–150*	Sets the threshold that the analog input must go above to set the virtual digital input high.
Low Threshold	0x3022	0x01–0x06	0.0–15.0 V *0–150*	Sets the threshold that the analog input must go below to set the virtual digital input low.

2.6 Processing and Interfacing Unit

A processing unit in an electric vehicle has to withstand an incredibly dynamic range of operation and runs even in the most rugged conditions. Interfacing the vehicular

sensors and processing data constantly for internal vehicular controls and external driver's display, to passenger's entertainment, the processing unit has to bear it all. With the electric vehicle integration, the unit has to interface with even more inputs and variables, such as handling conventional switches inputs on the dashboards to parsing and receiving the correct data for the electric vehicle controller. This turns looks to microcomputers which are a neat fit in terms of size, processing capabilities and interfacing capabilities.

Raspberry Pi, an established microcomputer in the market are one of the many low-cost computers which are open-sourced. Developed by Raspberry Pi Foundation to promote affordable teaching in schools, the base operating systems is Linux-based until recently, supports Windows 10 IoT Core. A few variants are also released, with some of them compared in Table 2.3. They become increasingly powerful and offers more interfacing options, not mentioning the increasing number of users and projects resources available on the internet.

Table 2.3: Differences between the Various Raspberry Pi Models

	Raspberry Pi 1 Model A	Raspberry Pi 1 Model A+	Raspberry Pi 1 Model B	Raspberry Pi 1 Model B+	Raspberry Pi 2 Model B
CPU	700 MHz single-core				900 MHz quad-core ARM Cortex-A7
Memory	256 MB		512 MB		1 GB
USB ports	1		2	4	
Onboard network	None		10/100 Mbit/s Ethernet		
Low-level peripherals	8× GPIO	17× GPIO	8× GPIO	17	

Software wise, the official version of the operating system would be Raspbian. The software is great in working with sensors and devices, and libraries are readily available from the web communities for common devices. The programming of this microcomputer revolves primarily around general purpose programming languages such as Python, C, and C++. However, support for a higher programming language is also possible such as Perl, or even object-oriented programming such as Java and Ruby.

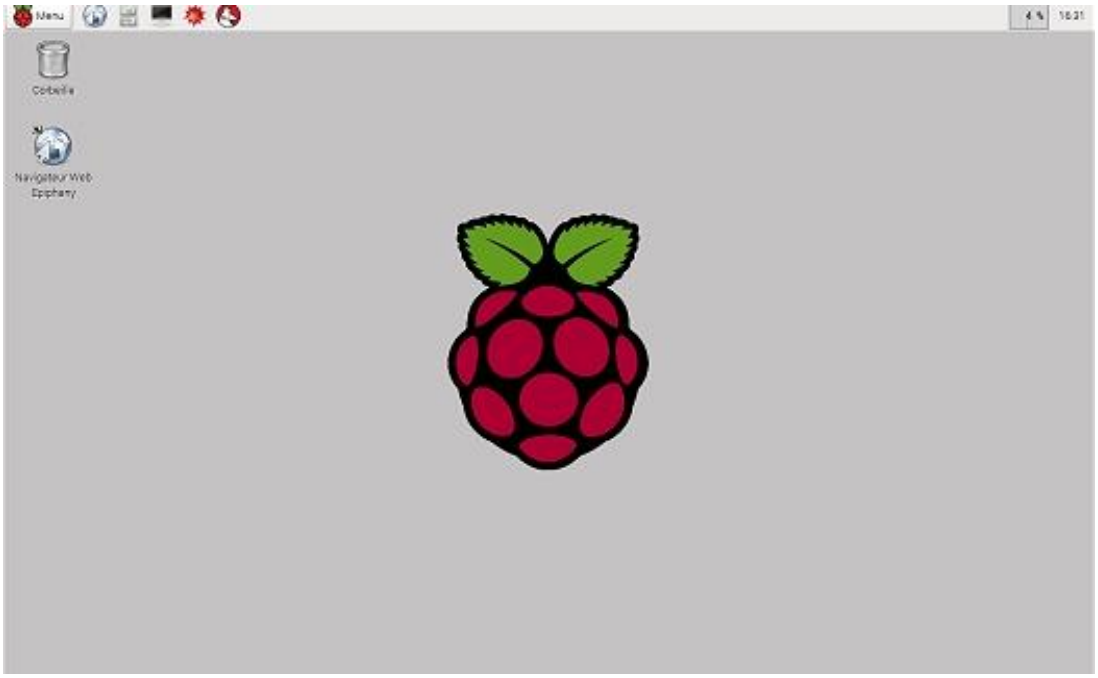


Figure 2.11: A Successfully Booted Up Raspbian OS with Raspberry Logo

With the capability of audio and video outputs, it is viable to integrate displays and sounds feedback to be directly driven from the Raspberry Pi platform. A graphical user interface can be built on top of the Desktop as in Figure 2.11, and provides additional support for an in-car entertainment system and even navigational systems. Besides, touch screen support and integration of sensors and modules will allow real-time vehicular information to be displayed and even configured on the fly. Data acquisition can also be done during the driving process, and further processed statistically to output either into the in-car display, or to be viewed on mobile or uploaded to the web. Such features allow the driver to have full access to the car information and controls and also be able to unleash the full potential of the vehicle.

CHAPTER 3

METHODOLOGY

3.1 Concept of EVICS

The EVICS revolves around the implementation of hardware and software on a scale from micro to macro components. For data acquisition, a combination of microcontrollers and Raspberry Pi is used. The channel of data source would include primarily from the electric vehicle's controller CANBUS network, and also feature sensors such as battery and temperature sensor modules, battery pack current sensor, and GPS module to be implemented into the EVICS system. This covers the whole range of data acquisition from conventional vehicle sources and also the electric vehicle parameters, which the data are relatively new to this electric conversion vehicle and has yet to have a place to process the new data.

Furthermore, feedback and actuation systems are also desired to improve the driving experience. The data retrieved from the vehicle can be processed and relevant details will be feedback to the user, which includes driving parameters, warning lights and beeping sounds. Display screens can be implemented not only to reveal vehicle details, but also allowing the user to interface the vehicular parameters and possibly changing them on the fly. A warning through the display and haptic feedback can also be provided via alert beeps to alert drivers on issues and emergencies. Besides, secondary systems and displays may also be integrated for passenger's ease of access on entertainment usage, or to accommodate driver's navigational system as well.

3.2 Vehicular Parameters Sensing

The different methods of obtaining the vehicular information are previously discussed in detail in Chapter 2. However, it is realized that existing sensors are already in place in the electric vehicle. The sensors include the individual battery pack voltage and temperature sensor and also a battery pack current sensor. In this case, the sensors will be utilized as well together with new sensing parameters to form an electric vehicular sensor network. This benefits as more parameters could be gathered from the electric vehicle itself, which allows much more comprehensive monitoring on the electric vehicle, however, the existing sensor compatibility will be an issue as the sensors were previously running via Mathworks MATLAB support package rather than the native Raspberry Pi operating system itself.

3.2.1 Existing Sensors Network

For the existing individual battery voltage and temperature sensors, it is based on Dallas 1-Wire protocol. The sensor chip is DS2436 which monitors both battery voltage and temperature. Its pin includes taking in the voltage input of the individual battery pack, and the surface of the chip itself acts as a temperature sensor. Therefore, the sensors circuit are closely mounted onto the batteries. An auxiliary power source is also required from the accessory battery to provide power to the op-amps and optoisolator, besides providing a reference voltage to the voltage measurement circuitry. The array of 12 individual battery sensors as in Figure 3.1 is then connected to a 1-wire master which handles the communication between the sensors and also converts the 1-wire signals from the sensors into serial protocol for Raspberry Pi. A USB to serial converter is also used which converts the serial protocol to USB to conserve the GPIO pins on the Raspberry Pi.

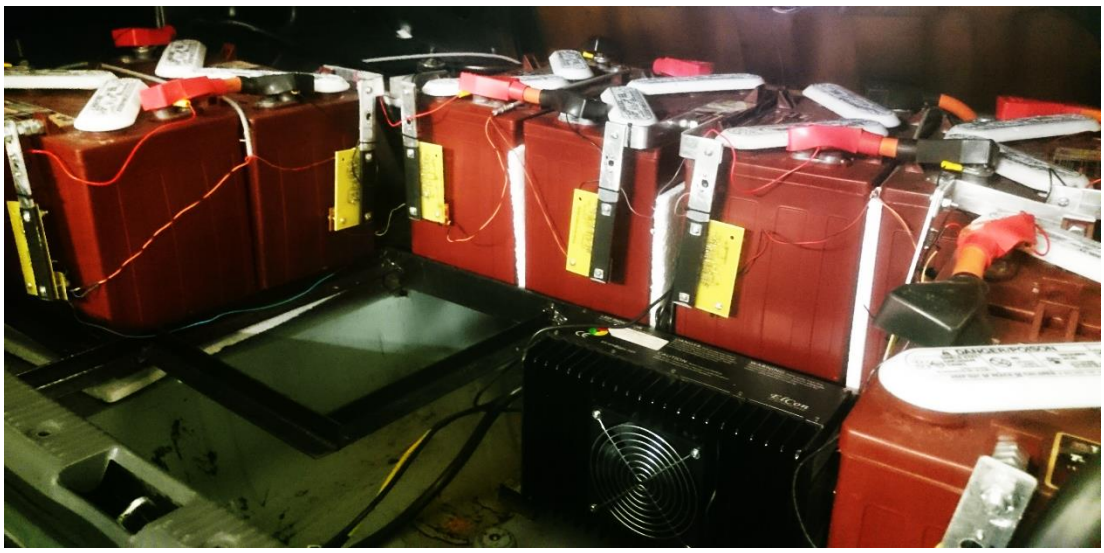


Figure 3.1: The existing individual battery sensor boards installed in the UTAR electric vehicle

Figure 3.2: The existing LEM current sensor installed in the UTAR electric vehicle

The battery pack current sensor present in place will be LEM Hall Effect Current Transducer as indicated in Figure 3.2. Capable of measuring current up to 600

A, the current sensor outputs in terms of analogous voltage. An ADC is then connected to the current sensor output before input into Raspberry Pi SPI pins.

3.2.2 Location of Electric Vehicle

GPS module is also installed into the sensors network to obtain the location of the vehicle. The GPS module has an antenna pin which allows the active external antenna to be connected to increase the gain up to 26dB. The external active antenna is connected and the antenna is rerouted in the vehicle cabin to place near the top of the windscreen to provide better reception of GPS signal. The attachment of the antenna block is as shown in Figure 3.3. The GPS module also utilizes serial communication. Therefore, a USB to serial converter is also used which converts the serial protocol to USB to conserve the GPIO pins on the Raspberry Pi.

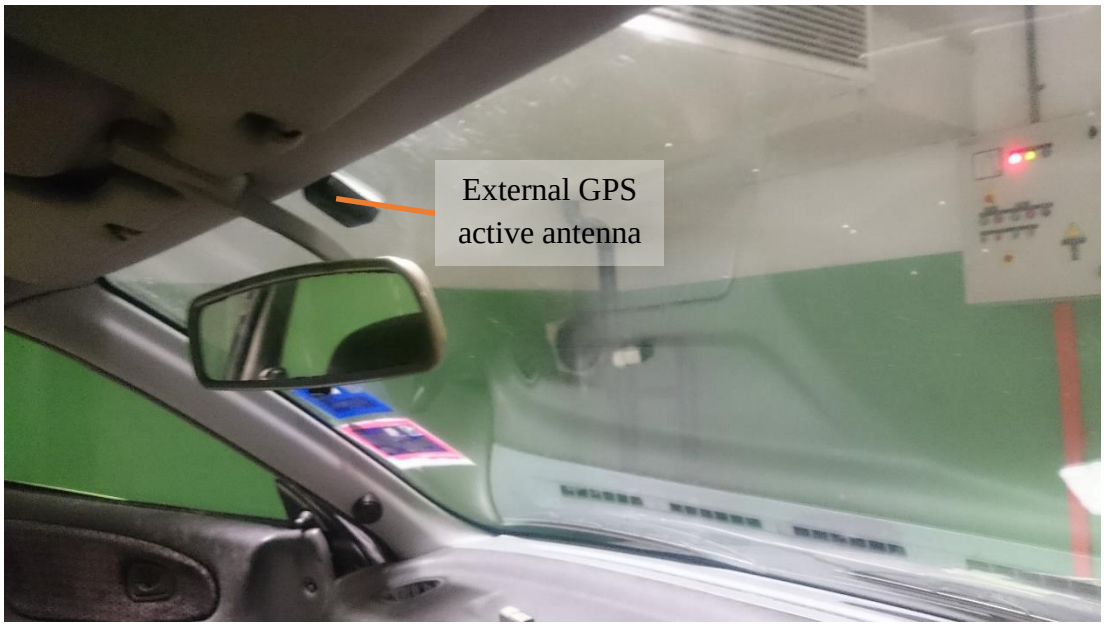


Figure 3.3: The external GPS antenna mounted on top centre of the windscreen

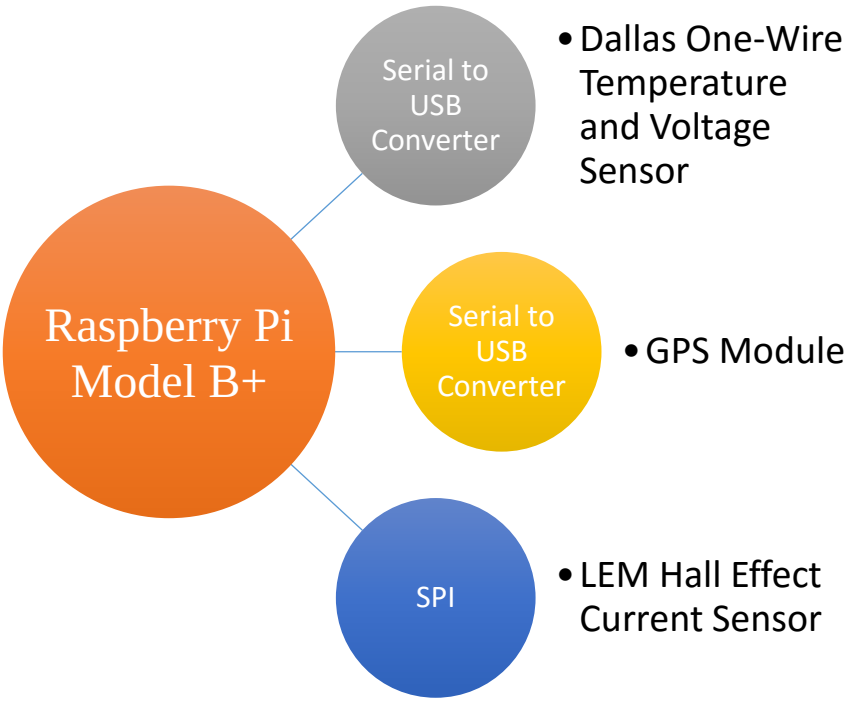


Figure 3.4: The block diagram showing the new setup of sensors network in the UTAR electric vehicle

3.2.3 CANBUS Circuit Setup

To acquire more comprehensive vehicular data in addition of external sensors in Figure 3.4, CANBUS method is preferred as the sensors are built right into the controller. As such, the data are deemed to be accurate and accuracy should be within manufacturer's tolerances as similar data is also used to manipulate the control systems of the vehicle.

At first, the CAN high and CAN low communication wires are identified from the Curtis 1239 electric vehicle controller. Pin 23, the CAN high wire and Pin 35, the CAN low wires are then connected to the controller's input and output connector, then running the wires through the vehicle firewall into vehicle cabin in a balanced, twisted pair configuration. The balanced line configuration is vital to ensure a stable 0V reference for all the receiving nodes in the line. This configuration works on the principle of current on the first signal line, is exactly opposite in current direction of the second signal line, effectively balances the line and avoiding crosstalk as well. This

reduces the interference susceptibility and radio frequency emission, which are critical to providing a stable 0V reference for data stability and increased bus communication speed.

With the wires ready, the next would be getting ready a CAN to USB interface. A circuit is then desired to convert the CAN signals into USB serial data for viewing and further computations. The device selection for such circuit will be Microchip's range of products, including MCP2551, MCP2515, with further integration with PIC18F14K50. Details of each device and the flow of data from CAN to USB are illustrated in Figure 3.5.

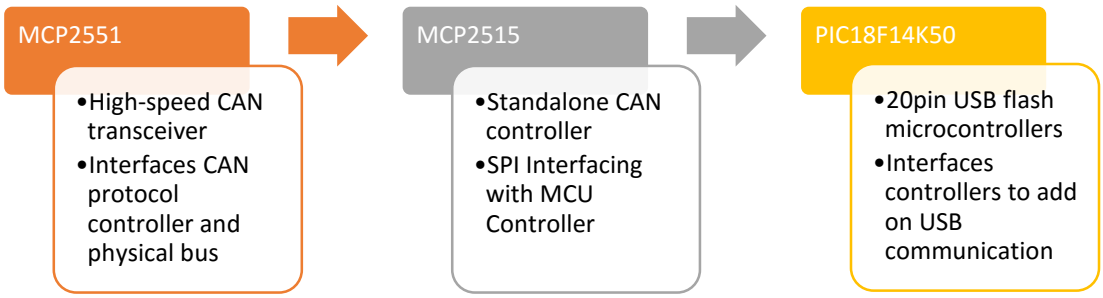


Figure 3.5: Basic overview of devices and flow of data communication between devices

The CAN messages will first travel into the MCP2551 CAN transceiver through the CAN high and CAN low ports of the device, as in Figure 3.6. The device is capable of transmitting and receiving CAN protocol controller and depending on vehicle controller's configuration, can operate up to baud rate of 1Mb/s. Being a node in the CAN system, it would be able to retrieve the raw information from the CAN network and further process the data into suitable signals for transmission, through its transmitting and receiving ports. Typically, electromagnetic interference, electrostatic discharge, electrical transients and radio frequency interference are also present in the CAN network, which may indirectly affect the performance of the CAN devices. Also, present working with automobiles that are constantly put in stressed and adverse conditions, raising the susceptibility of faults such as battery short-circuit and excessive current loading. Protection of the device and its output and input ports to and

from other devices must be accounted. Thus, with the presence of this specialized CAN transceiver, various voltage spikes and issues related to interference can also be minimized or eliminated with the presence of this device as an isolation between the CAN network and the CAN controller. It is also noted that the choice of the temperature range for this chip is chosen at industrial grade, which ranges from -40°C to 85°C which is considerably sufficient for this project.

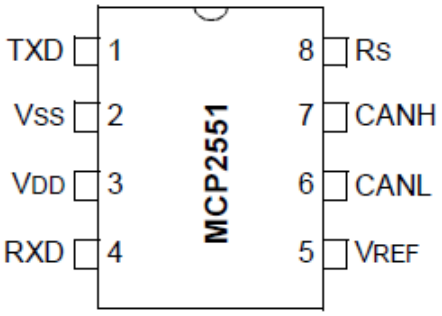


Figure 3.6: Input and Output Ports of High-Speed CAN Transceiver MCP2551 (Microchip, 2003)

From the connections of transmitting and receiving on the MCP2551, the CAN messages which are clean are then provided to transmit and receive ports of MCP 2515. This specialized CAN controller comes equipped with SPI interface to allow further interfacing with microcontrollers. The main blocks of the device are shown in Figure 3.7, consisting CAN modules, SPI interface blocks, and controlling logic registers. Transmission and reception of all messages are handled by CAN module, then loaded into control registers, and further initiated by controls for SPI transmission of data. Control logic interfaces the other two blocks, to allow interrupts, status registers access and manually initiated transmission of data, all possibly accomplished and accessed via SPI interface. The SPI protocol block allows read-write operations into all registers via generic read-write commands and specialized SPI commands. This device also spots oscillator input and output which allow a single oscillator control for the circuit and its overall circuit controls, simplifying the bit timing issues across different devices, eliminating the needs of readjustment on baud rate pre-scaler on each individual devices. The choice of the temperature range for this chip is also chosen at industrial grade, similar to the temperature range chosen for MCP2515.

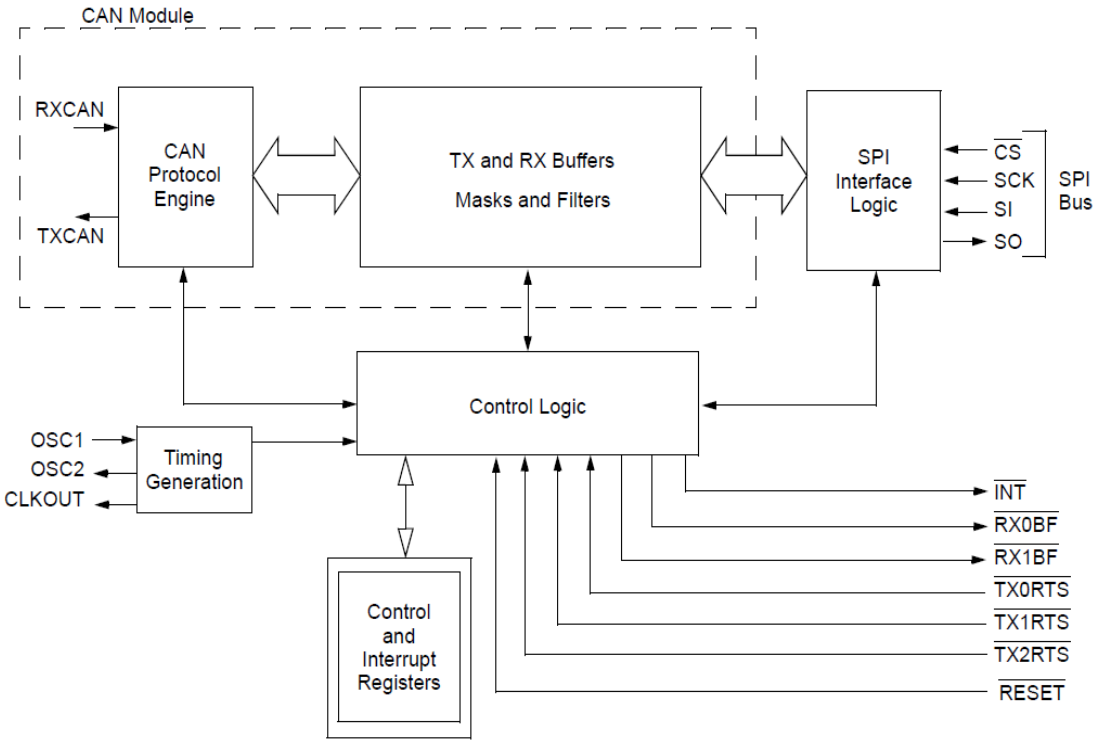


Figure 3.7: Device Overview of MCP2515 (Microchip, 2007)

The connections into the microcontroller from the CAN controller basically consists of SPI interfacing connections, interrupts and oscillator connections. Further connections introduced to the microcontroller includes USB interfacing, programming jumper, voltage smoothening circuit and a status indicator LED. Taking advantage of its embedded USB capability and its wide implementation on SPI and even UART, the device is chosen. It also runs off 5V which is the USB voltage, easily powering the device without the requirement of external power supply. This allows simple USB communication to be established and to be output into devices for data logging or further processing. The overall pinouts are shown in Figure 3.8.

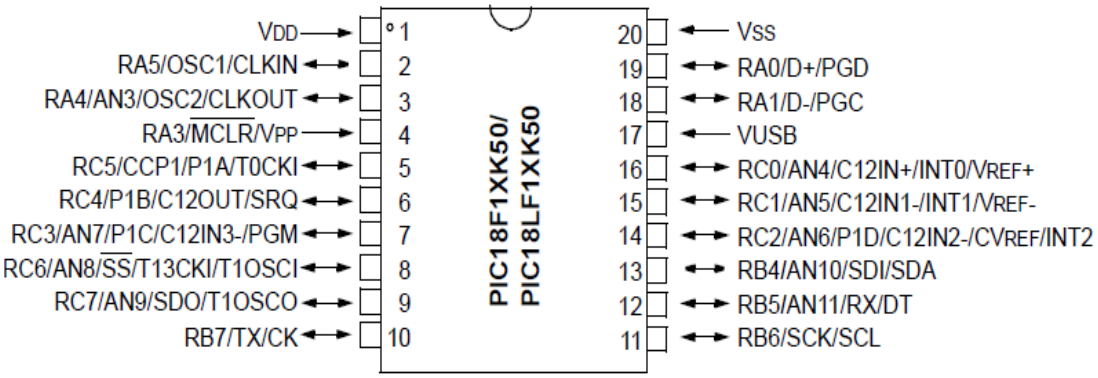


Figure 3.8: Overview of Pinouts on PIC18F14K50 (Microchip, 2009)

The complete circuit connections for the CAN to USB interface is as indicated in Figure 3.9. Also in Figure 3.10, the constructed working circuit is as shown. The connections on the CAN transceiver MCP2551 and CAN controller MCP2515 are rather self-explanatory where minimum circuit connections are connected. For the microcontroller PIC18F14K50, output pins for PGC and PGD are connected to USB for programming and communication purposes. Also, a jumper pin of JP1 is used for setting the circuit into bootloader mode once sunk to the ground, allowing programming to be performed. Meanwhile, JP3 is reserved for CAN termination purposes, which is selectable for vehicles without termination for transmission over much longer bus wires at higher speed. JP4 would be connected directly to the CAN high and CAN low of the vehicle wires respectively.

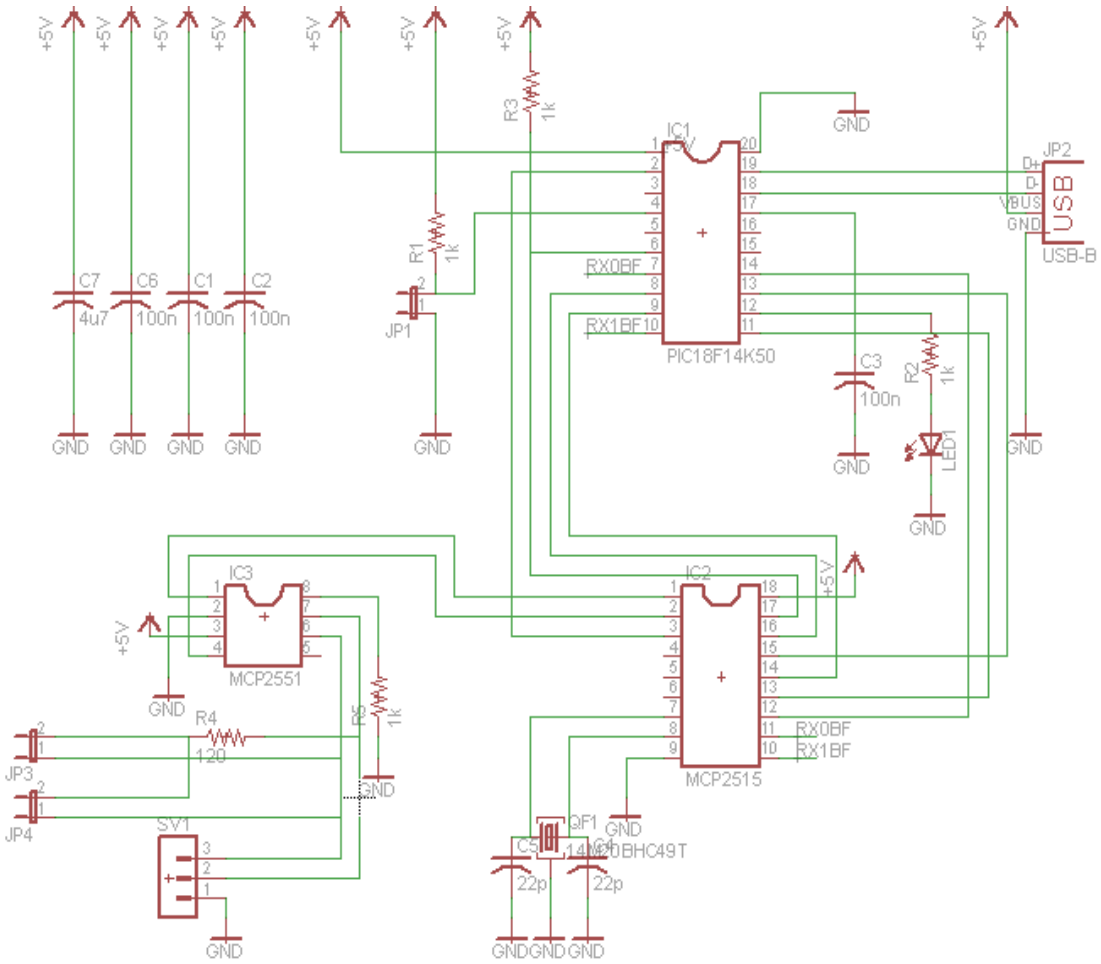


Figure 3.9: Overall Circuit Configuration for CAN to USB Interface

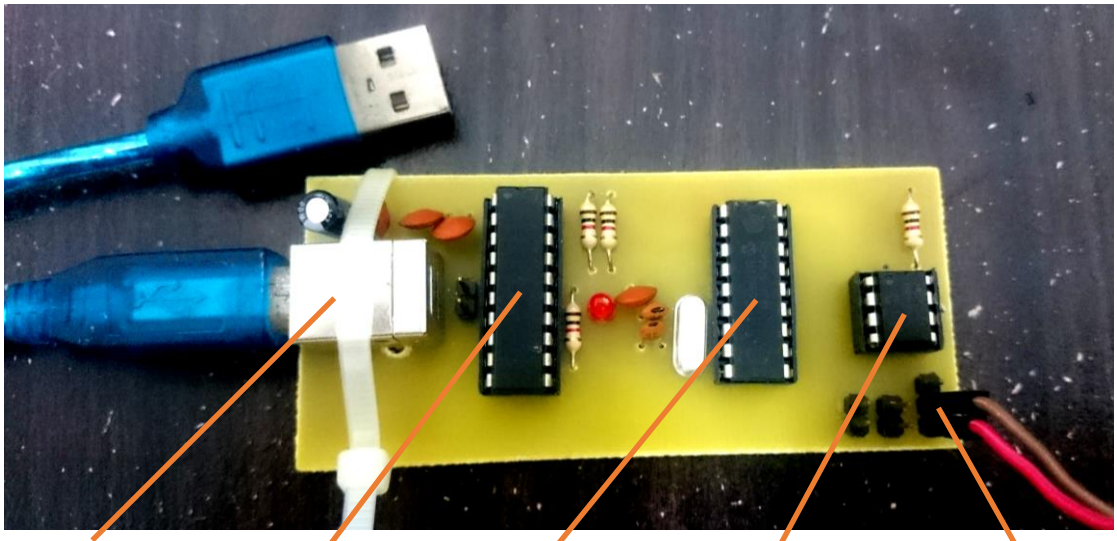


Figure 3.10: Fabricated Circuit Board for CAN to USB Interface

3.2.4 Data Acquisition and Processing

It is noted that the CAN network messages are producing only raw data that is not ready for direct viewing and interpretation. Further data analysing is required in order to transform the data into meaningful information for the driver to read easily.

As in Figure 3.11, the table indicates the generic CAN messages that are directly obtainable from the Curtis electric vehicle controller. With the default addresses stated at 0x601h and 0x602h for parameters transmission, the messages can be obtained directly with the CAN to USB interface. The parameters obtainable are also as shown in the Figure 3.11.

Such promising data which are directly acquired from the controller would need to be further analysed and interpreted before it became a meaningful data to be observed by the driver. However, it is to note that some data are available but not retrieved due to unconnected inputs, which the input is optional to this electric vehicle conversion.

Generic CAN Messages from Controller

	ADDRESS ID						
	CAN ADDRESS 0x601	Units	Scale		**CAN ADDRESS 0x602**	Units	Scale
Byte0	Motor RPM high byte	RPM	1		Stator Frequency high byte	Hz	1
Byte1	Motor RPM low byte				Stator Frequency low byte		
Byte2	Motor Temp	Deg C	-40 to 200		Controller Fault Primary		
Byte3	Controller Temp				Controller Fault Secondary		
Byte4	RMS Current high byte	Amps	0.1		Throttle Input	%	1
Byte5	RMS Current low byte				Brake Input		
Byte6	Capacitor Voltage high byte	Volts	0.1		System Bits*		
Byte7	Capacitor Voltage low byte				Not used		

Figure 3.11: Generic CAN Messages Acquirable Directly from the Controller (HPEVS, 2014)

3.2.4.1 Revolution per Minute (RPM)

Obtainable under CAN address 0x601, the byte zero and byte one of the address represents the electric vehicle motor's RPM. Also indicated is the value is in the scale of one. The calculation required to obtain the RPM would be as in formula (3.1), with reference to Table 3.1.

$$RPM = (H_H \times 16^3 + H_L \times 16^2 + L_H \times 16^1 + L_L \times 16^0) \quad (3.1)$$

Where

H_H = 4 higher data bits of byte 0 of address 0x601h expressed in hexadecimal form

H_L = 4 lower data bits of byte 0 of address 0x601h expressed in hexadecimal form

L_H = 4 higher data bits of byte 1 of address 0x601h expressed in hexadecimal form

L_L = 4 lower data bits of byte 1 of address 0x601h expressed in hexadecimal form

Table 3.1: Hexadecimal to Decimal Conversion Table

Hex base 16	**Decimal base 10**
0	0
1	1
2	2
3	3
4	4
5	5
6	6
7	7
8	8
9	9
A	10
B	11
C	12
D	13
E	14
F	15

3.2.4.2 Motor Temperature

Obtainable under CAN address 0x601, the byte two of the address represents the electric motor's temperature in degree Celsius. Also indicated is the value can range from -40 to 200. The calculation required to obtain the motor temperature would be as in formula (3.2), with reference to Table 3.1.

$$Motor\ Temperature\ (°C) = (M_H \times 16^1 + M_L \times 16^0) \quad (3.2)$$

Where

M_H = 4 higher data bits of byte 2 of address 0x601h expressed in hexadecimal form

M_L = 4 lower data bits of byte 2 of address 0x601h expressed in hexadecimal form

3.2.4.3 Controller Temperature

Obtainable under CAN address 0x601, the byte three of the address represents the electric motor's temperature in degree Celsius. Same as motor's temperature, the values indicated will range from -40 to 200. The calculation required to obtain the motor temperature would be as in formula (3.3), with reference to Table 3.1.

$$Controller\ Temperature\ (°C) = (C_H \times 16^1 + C_L \times 16^0) \quad (3.3)$$

Where

C_H = 4 higher data bits of byte 3 of address 0x601h expressed in hexadecimal form

C_L = 4 lower data bits of byte 3 of address 0x601h expressed in hexadecimal form

3.2.4.4 Motor Current

Obtainable under CAN address 0x601h, the byte four and byte five of the address represents the electric motor's current in Amperage, on the scale of 0.1. Formula (3.4) reveals the calculation required to obtain the motor current, with reference to Table 3.1.

$$Motor\ Current\ (A) = (H_H \times 16^3 + H_L \times 16^2 + L_H \times 16^1 + L_L \times 16^0) \times 0.1 \quad (3.4)$$

Where
H_H = 4 higher data bits of byte 4 of address 0x601h expressed in hexadecimal form
H_L = 4 lower data bits of byte 4 of address 0x601h expressed in hexadecimal form
L_H = 4 higher data bits of byte 5 of address 0x601h expressed in hexadecimal form
L_L = 4 lower data bits of byte 5 of address 0x601h expressed in hexadecimal form

3.2.4.5 Battery Voltage

Byte six and byte seven of the address 0x601h represents the battery pack voltage in Volts, on the scale of 0.1. Formula (3.5) reveals the calculation required to obtain the battery pack voltage, with reference to Table 3.1.

$$Battery\ Voltage\ (V) = (H_H \times 16^3 + H_L \times 16^2 + L_H \times 16^1 + L_L \times 16^0) \times 0.1 \quad (3.5)$$

Where
H_H = 4 higher data bits of byte 6 of address 0x601h expressed in hexadecimal form
H_L = 4 lower data bits of byte 6 of address 0x601h expressed in hexadecimal form
L_H = 4 higher data bits of byte 7 of address 0x601h expressed in hexadecimal form
L_L = 4 lower data bits of byte 7 of address 0x601h expressed in hexadecimal form

3.2.4.6 Controller's Fault Monitoring

Moving to address 0x602h, byte two and byte three represents the controller's fault codes on each controller's respectively. Byte two represents the primary controller fault value and byte 3 represents secondary controller fault code. Formula (3.6) reveals the calculation required to obtain the controller's fault code for the primary controller, with reference to Table 3.1. The fault codes then can be compared against manufacturer's fault codes to gain further insights on the problem arose. An example of the fault codes is shown in Table 3.2. The formula 3.6 is also applicable on the secondary controller with the byte changes to byte 3.

$$Fault\ Code = (F_H \times 16^1 + F_L \times 16^0) \tag{3.6}$$

Where

F_H = 4 higher data bits of byte 2 of address 0x602h expressed in hexadecimal form

F_L = 4 lower data bits of byte 2 of address 0x602h expressed in hexadecimal form

Table 3.2: Part of the Fault Codes from Manufacturer's Manual (HPEVS, n.d)

Code	Programmer Display (Effect or Fault)	Possible Cause	Set/Clear Conditions
12	Controller Overcurrent	1) External short of phase U, V, or W motor connections 2) Motor parameters are mis-tuned 3) Controller defective	**Set:** Phase current exceeded the current measurement limit **Clear:** Cycle KSI
13	Current Sensor Fault	1) Leakage to vehicle frame from phase U, V, or W (short in motor stator) 2) Controller defective	**Set**: Controller current sensors have invalid reading **Clear**: Cycle KSI
14	Precharge Failed	1) External load on capacitor bank (B+ connection terminal) that prevents the capacitor bank from charging 2) See Monitor menu >> Battery: Capacitor Voltage	**Set**: Precharge failed to charge the capacitor bank to KSI voltage **Clear**: Cycle Interlock input or use VCL function *Precharge()*
15	Controller Severe Undertemp	1) See Monitor menu >> Controller: Temperature 2) Controller is operating in an extreme environment	**Set**: Heatsink temperature below -40° C **Clear**: Bring heatsink temperature above -40°C, and cycle interlock or KSI
16	Controller Severe Overtemp	1) See Monitor menu >> Controller: Temperature 2) Controller is operating in an extreme environment 3)Excessive load on vehicle 4) Improper mounting of controller	**Set**: Heatsink temperature above +95°C **Clear**: Bring heatsink temperature below +95°C, and cycle interlock or KSI

3.2.4.7 Throttle Input Sensing

Throttle input sensing is represented by byte four of the address 0x602h, in terms of percentage. The calculation required to obtain the throttle input percentage would be as in formula (3.7), with reference to Table 3.1.

$$Throttle\ Input\ (\%) = (T_H \times 16^1 + T_L \times 16^0) \quad (3.7)$$

Where

T_H = 4 higher data bits of byte 4 of address 0x602h expressed in hexadecimal form

T_L = 4 lower data bits of byte 4 of address 0x602h expressed in hexadecimal form

3.2.4.8 Systems Bits on State of Electric Vehicle

Byte six of the address 0x602h represents the systems bits on the current state of the electric vehicle. The system bits that are applicable are revealed as in Table 3.3.

Table 3.3: System Bits Output Configuration

Bit	Logic
0	Economy bit
1	Regenerative bit
2	Reverse bit
3	Brake Light bit

3.2.5 Data Controls and Interfacing

For drivers and users, the CAN to USB interface can be connected to Raspberry Pi to allow data transfer and processing to produce values for driving purposes. This involves setting up the microcomputer to install modules for the CAN transceiver and controller and initializing SPI drivers. With proper configuration, one would be able

to retrieve the raw data and processing it before outputting into the dash for drivers to monitor the driving parameters. The setup involves mainly only on reading of CAN network data. Logging of data is also performed with scripts using Python programming.

In this project, it is also noted that certain CANBUS parameters that are unable to be physically generated by the electric vehicle controller. This includes abstract cases such as fault code and system bit generation. However, such issue can be overcome by using CANBUS message simulator to send out the CANBUS messages. The USBTinViewer, running on Windows platform is used mainly to generate the CANBUS data bits onto a CANBUS board as mentioned in Section 3.2.3, and another separate CANBUS board is used as a receiver to receive messages to be processed by Raspberry Pi. The hardware setup is shown as in Figure 3.12. Software wise, the setup is indicated in Figure 3.13.

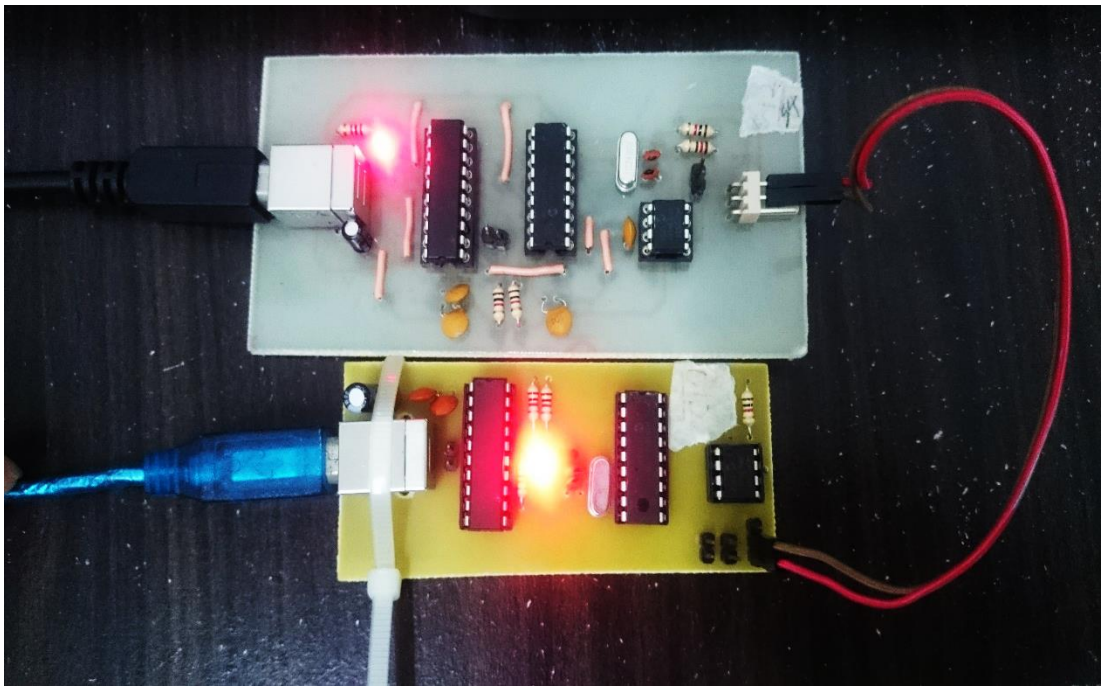


Figure 3.12: CANBUS messages simulation using two CANBUS boards

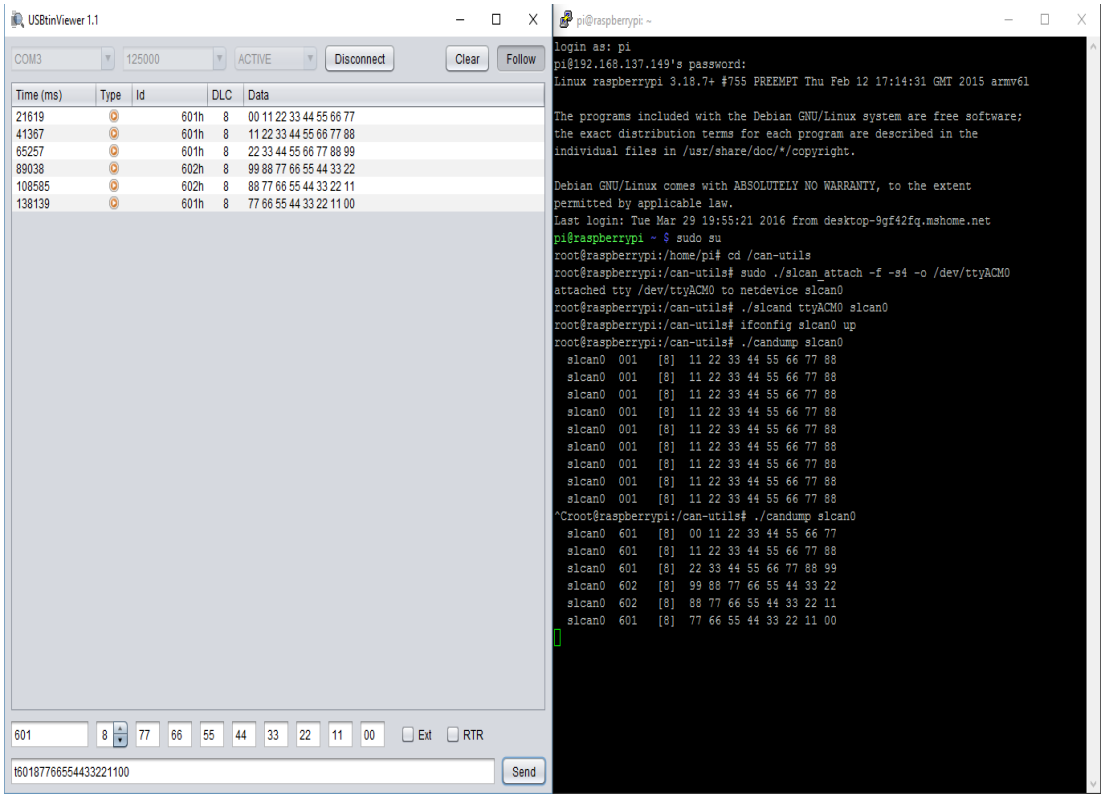


Figure 3.13: USBTinViewer software generating CANBUS messages on the left window, while Terminal in Raspberry Pi receiving the messages on the right window

3.3 Vehicular System Network integration with Electric Vehicle

In this project, the sensors networks, modules, and data acquisition system is also a step further integrated into the electric vehicle. As opposed to the approaches done beforehand, where the user needs to manually start the system and makes configuration before obtaining the data, the system is now able to run independently without user intervention, yet will be up and ready before the user drives the car, and properly shut down itself when the car is off. The data logs can also be easily accessed through the plugged micro SD card. The information can then be accessed from a personal computer running Linux distribution for checking and analysis.

3.3.1 Changing Raspberry Pi and Sensors Network Power State using Switch

To allow Raspberry Pi to auto start-up and shut down, a number of Raspberry Pi pins and GPIO would be required. The bootup pins are first identified on the Raspberry Pi used. Its location is the RUN pins of the Raspberry Pi board, as indicated in Figure 3.14. Pin headers are then soldered to allow ease of connections to output.

During normal operation, the IC pinout of D15 is held high as the circuit is driven by pull-up resistor if R15 as shown in Figure 3.15. As long as the pinout of D15 RUN is in high logic state, the Raspberry Pi operations are normal. However, once the pins of P6 is shorted, the RUN pin is pulled to ground, the state of RUN pin is cleared. Thus driving the operation of Raspberry Pi into a reset situation. If Raspberry Pi is powered on, the system will then soft reboot as the reset signal is triggered. In this case of usage where the state of Raspberry Pi is in off mode but power is applied, the system will then reset, leading to Raspberry Pi being turn on from its off state. Therefore, installing a switch that shorts between the pins of D6 will trigger the Raspberry Pi to on state. A Python script is also written and executed as root user to enable Python programs such as sensors network programs and soft shut down programs to autostart upon completion of booting of the Raspberry Pi system.

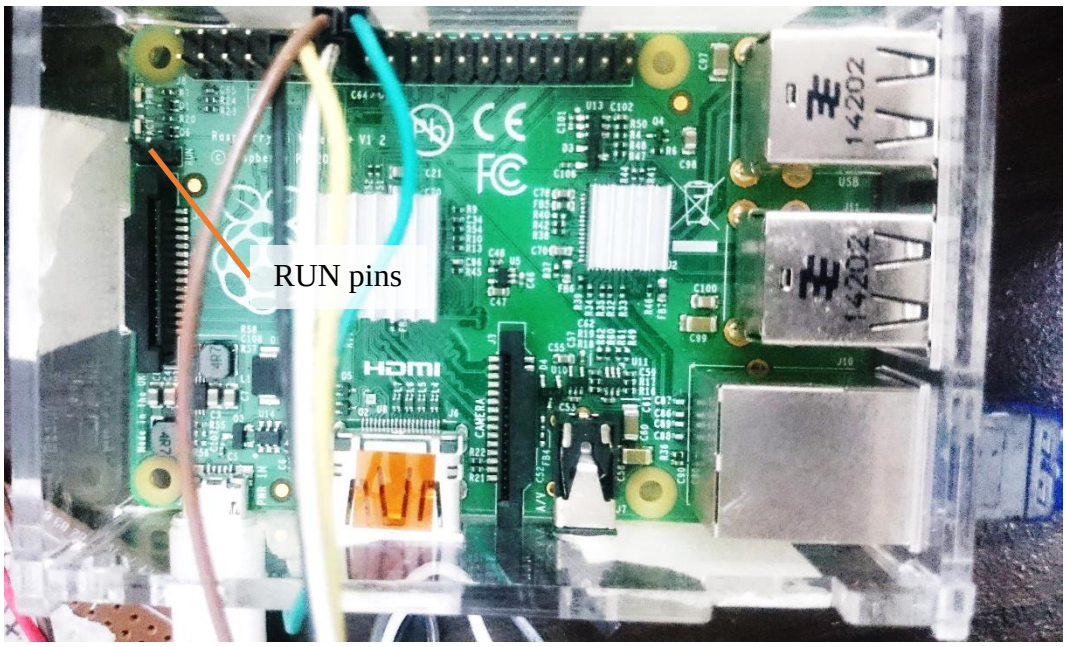


Figure 3.14: The RUN pins soldered with pin headers

It is noted that if the power to the Raspberry Pi system is cut abruptly, it may lead to system corruption, data loss and subsequently, boot failure, Therefore, a soft shutdown is preferred. For the Raspberry Pi system, one of the ways utilized is by adding a switch to its GPIO, and providing a negative trigger to result in Raspberry Pi executing shut down the process. By using GPIO 23 or physical pin of pin 16 a switch can be connected to the GPIO 23 and ground. A Python script, which has been autostarted during boot as the root user, would then provide the command for the system to shut down once the pin sensed the falling edge of the negative trigger. As of Figure 3.16, the auto shut down Python script ran by the system would be based on interrupt and not polling basis. Thus, would not burden the system precious processing power during running operation of Raspberry Pi, where the system is expected to run scripts of various sensors network.

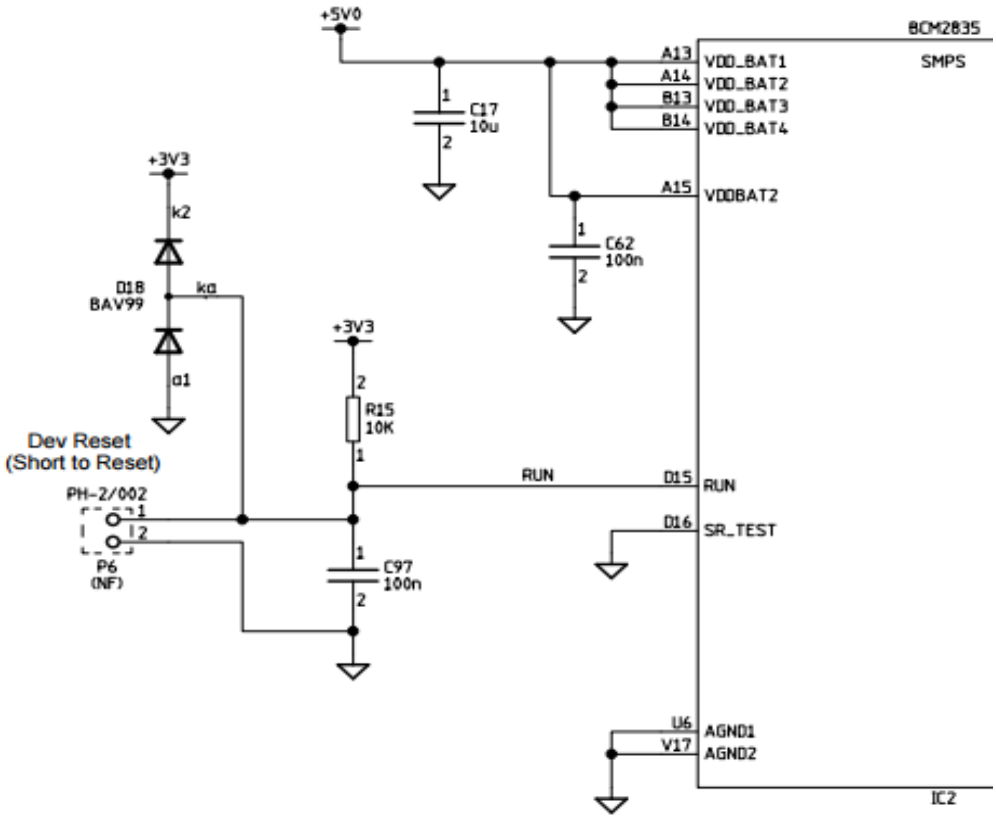


Figure 3.15: Datasheet of Raspberry Pi Model B+ showing the schematic of RUN pins to the IC pinout

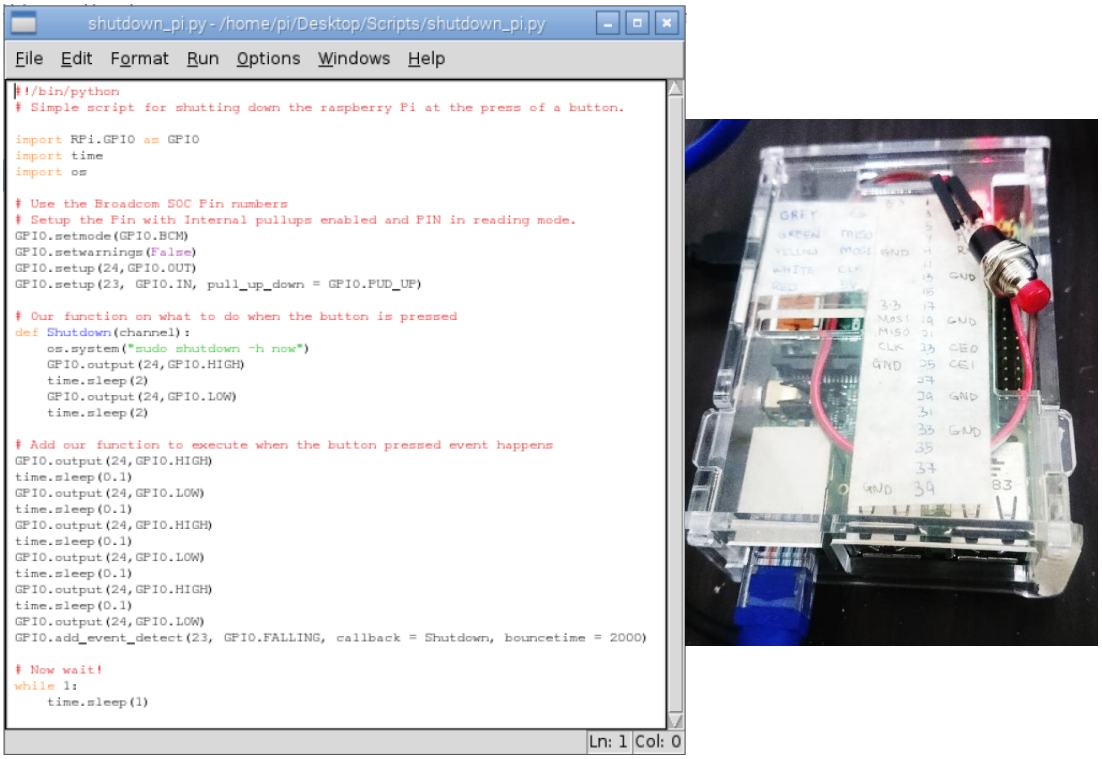


Figure 3.16: Testing of shutdown button using a switch with GPIO 23 or pin 16 of Raspberry Pi. The Python program is indicated on the left.

3.3.2 Raspberry Pi and Sensors Network Auto Power with Vehicle State

With the successful circuit and programming configurations for shutdown and booting, the Raspberry Pi and sensors network are then configured to be auto powered on and off depending on vehicle state. The desired state of on would be prior to vehicle start up as the Raspberry Pi and sensors network require some time to boot up before they can be online and information will only be available to the driver after that. For off state, it would be best if it can be triggered off and shut down properly even after the key is taken off from the car for the driver's convenience. Therefore, there arise a need of a soft shut down even after power from the key is removed.

To address such issue, an innovative solution is proposed by utilizing the connections in the existing alarm system of the electric vehicle. The modern alarm system supports sensors input, which provides power to the sensors once the car is

armed, and cuts off power when the car is unarmed. Thus, by utilizing the power leads of the sensor input as in Figure 3.17, a circuit can be built to trigger on the Raspberry Pi upon alarm unarming and trigger shutdown the Raspberry Pi upon arming. The circuit configuration is as shown in Figure 3.18.

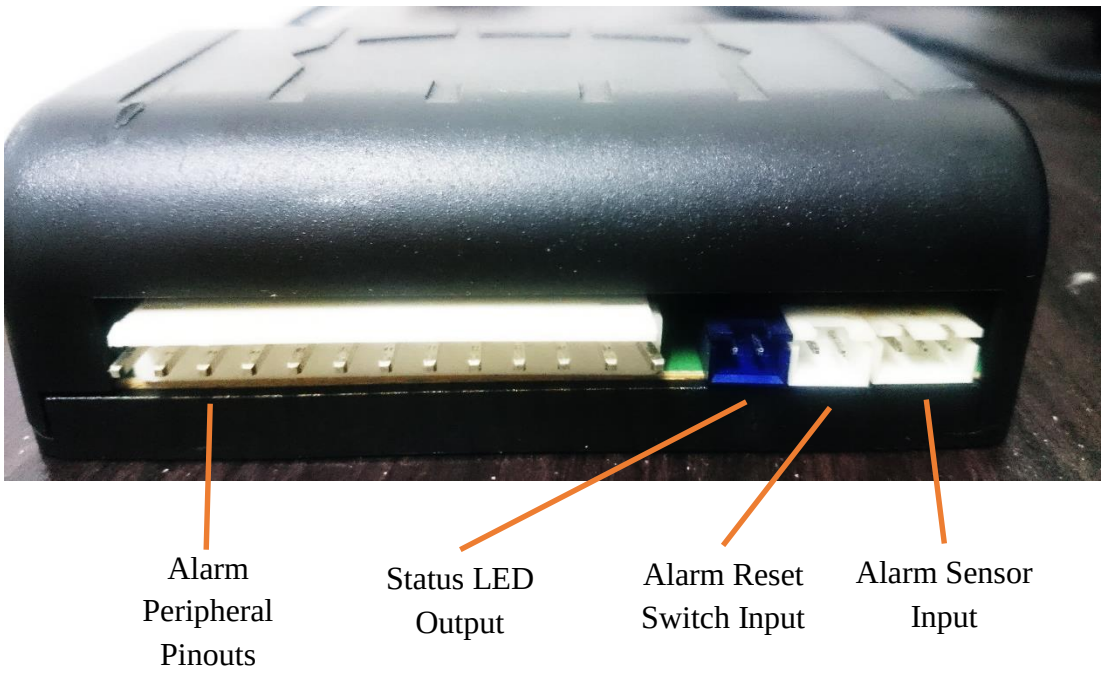


Figure 3.17: Alarm module overall pinouts, the alarm sensor input pins are used for auto powering the Raspberry Pi and its sensors network

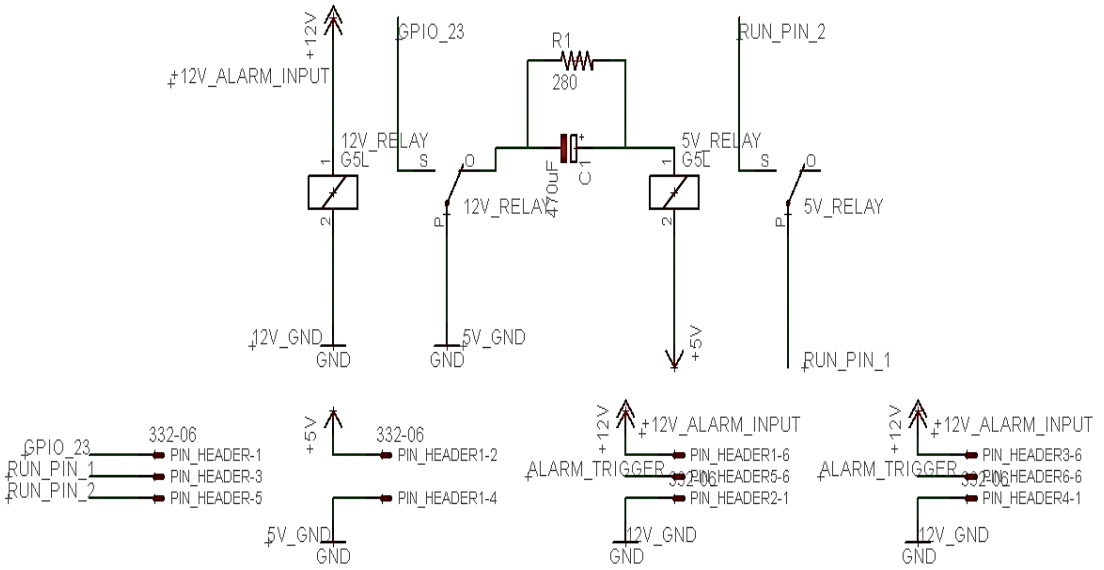


Figure 3.18: The circuit connections of between the alarm module and Raspberry Pi to provide auto-booting and shut down

The circuit mainly consists of 2 relays with several pin headers serves as outputs and inputs between the Raspberry Pi and the alarm module. An extra pin header is also provided to allow alarm sensors to be connected. During arming of the alarm, the alarm module sensor input pin is high, thus, provide 12 V for the 12 V relay to latch. The latching of the relay closes the normally open connection of 5V ground and GPIO 23. Thus triggering shut down of the Raspberry Pi as the background auto shut down Python script is waiting for negative edge trigger to power off the system.

When the driver unarms the alarm, the alarm module ceases to provide 12 V output to the sensor input. This causes the 12 V relay to unlatch and the pole of the relay is thrown to normally closed pins. The connection of the 5 V ground causes the 5 V relay on the right to be latched. Latching of the relay then shorts the RUN pins of the Raspberry Pi, thus triggering booting of the Raspberry Pi.

It is noted that as the RUN pins are to be closed momentarily only to avoid power cycle of Raspberry Pi. Therefore, a simple circuit consisting of a capacitor and a resistor is constructed to convert the toggle to momentary latching of the relay to provide the appropriate triggering signal. The toggle to momentary circuit works with the principle of DC charging characteristics of a capacitor. As the capacitor receives DC power from the completion of the circuit, the capacitor charges. Thus allowing the charge current to flow through the relay windings, energizing the relay, and shorts the RUN pins of the Raspberry Pi. However, once the capacitor is fully charged, the charging current stops and de-energizes the relay. The RUN pins are then open-circuited although the relay circuit still receives DC power from the 12 V relay. A 280 Ω resistor is provided to discharge the capacitor quickly once the 12 V relay is de-latched.

3.3.3 Electric Vehicle System State Output

Also installed is a simplified yet essential vehicle instrument cluster systems to provide physical output for the electric vehicular parameters. One of the outputs that are

available for an electric vehicle as oppose to the conventional vehicle will be the status LED for the system state of the electric vehicle. Since the electric vehicle is capable of running in normal mode, economy mode, and regenerative mode, it would be preferable if the driver can opt to know the current system state. The system state information would be obtained from CANBUS. An array of LEDs is installed by means of integration into existing dashboard to allow the driver to be conscious of the current vehicle system state. This includes an orange coloured LED for economical driving, a green coloured LED for regenerative braking state and a red coloured LED reserved for system fault. As during normal cruising, all the LEDs will remain off.

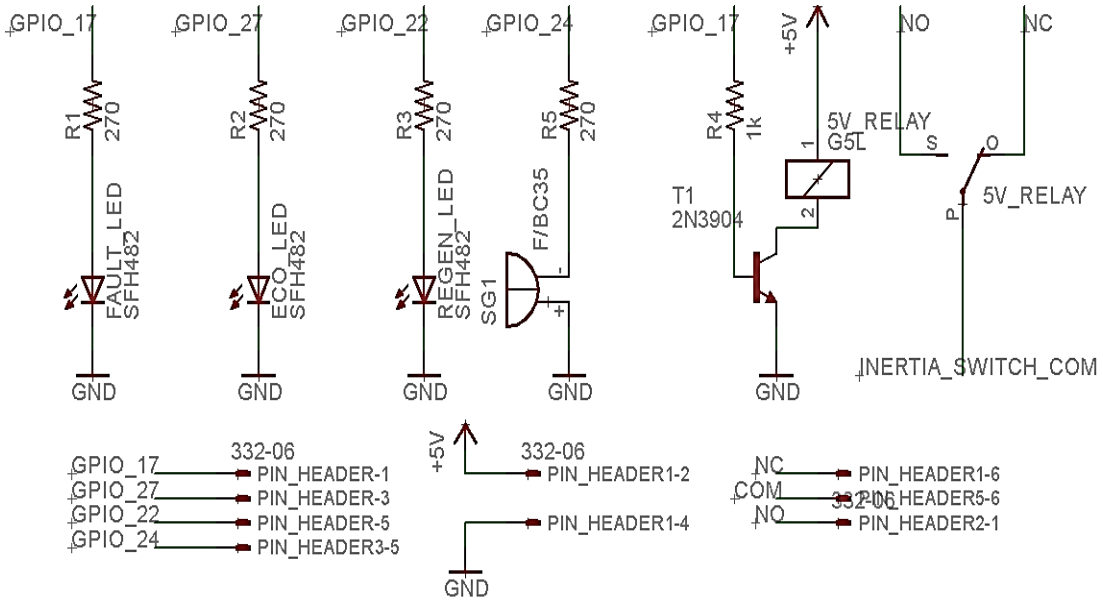


Figure 3.19: The circuit connections of the electric vehicle system's output state which provide driver's convenience on checking the vehicle system state

Figure 3.19 reveals the system state output circuit setup. From the program, GPIO 17 output is configured for fault status output, while GPIO 22, GPIO 23 and GPIO 24 are configured outputs for economy status, regenerative braking status, and system status respectively. Each of the GPIO outputs is respectively connected to a resistor and an LED for indication, and GPIO 24 connected to a buzzer. For the fault status GPIO, an extra relay is connected in parallel to the GPIO output. The relay is connected in the configuration with a relay driver using an NPN bipolar junction transistor to allow the relay to latch properly using Raspberry Pi GPIO limited 3.3 V

output. The circuit is connected in series with the inertia switch of the vehicle, on the 5 V relay with a common pin and normally closed pin. Such connection in series with the inertia switch is the preferable point as the inertia switch also provides a similar cut-off function in the case of emergencies such as collision. This is indicated as of Figure 3.20. For the circuitry, upon triggering of fault, the 5 V relay would be latched to normally open pin, opening the circuit of the 12 V main contactor of the electric vehicle controller, which is will then cut off the contacting with 144 V battery pack source during emergencies. The buzzer would also sound as an indication of faults.

The circuits in Section 3.3 are finally combined all together into the one-piece board as shown in Figure 3.21.

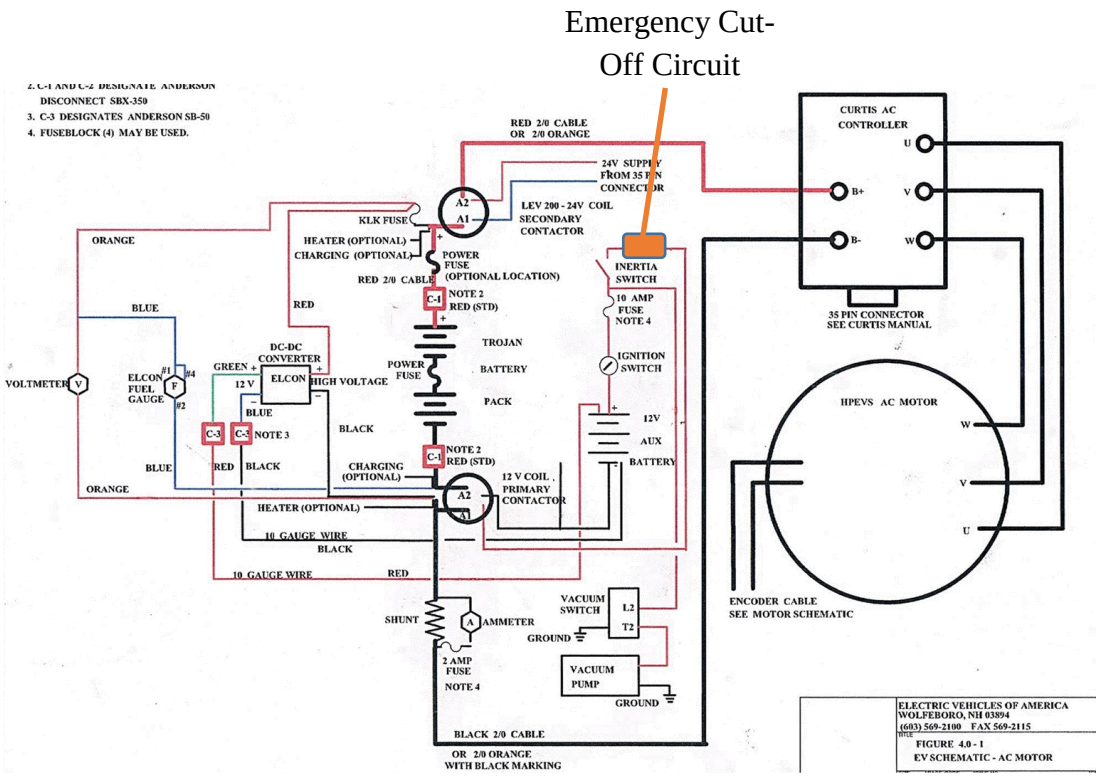


Figure 3.20: The schematic showing the electric vehicle wiring diagram. The emergency cut-off circuit is placed in series with inertia switch as a safe cut-off point

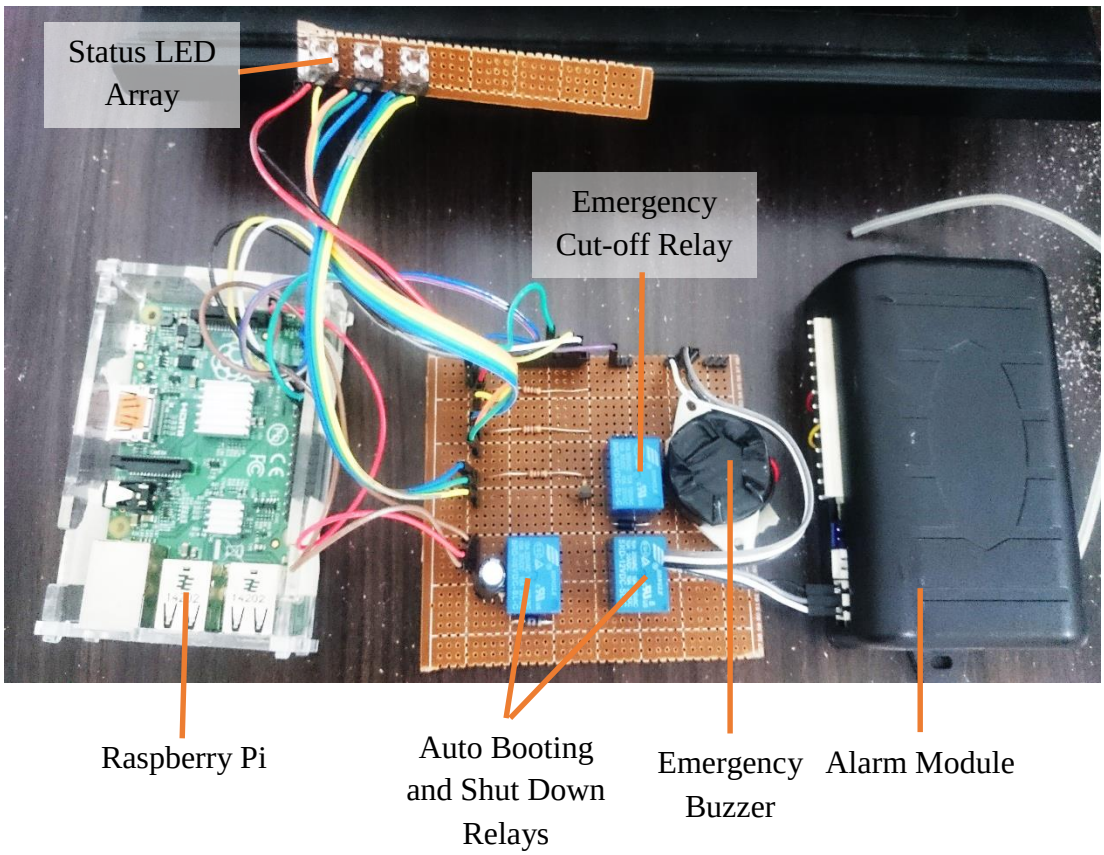


Figure 3.21: The circuits of Raspberry Pi auto powering and electric vehicle system state output

3.4 Python Programming and Libraries

In this project, Raspberry Pi loaded with Raspbian operating system is used. To allow interfacing with the sensors networks and data processing, Python programming language is used. The process of data acquisition from the sensors network is via logging of data obtained from the sensors network into a separate CSV file, written into the micro SD card plugged into the Raspberry Pi.

For the individual battery pack voltage and temperature sensor, the PySerial module is used as the communication methods are of serial basis. Further communication with the sensors masters and slave are of meticulous polling of sensors ID and bits. The GPS module, however, requires both the Python modules of PySerial and GPSd as it also requires GPS drivers to run. Meanwhile, the current sensor utilizes

Spidev module for SPI access. For CANBUS data acquisition, the Python program runs native Linux bash script as a subprocess, since Python modules are not readily supported. Therefore, the Raspbian is custom installed with CANBUS networking and SPI kernels and drivers, which are recompiled in Linux Ubuntu on a personal computer. Then, CAN-utils of SocketCAN package is also compiled and installed to enable successful data acquisition of CANBUS.

CHAPTER 4

RESULTS AND DISCUSSION

4.1.1 CANBUS Messages

With reference to the controller's CANBUS data as in Figure 3.10, they are in the form of two addresses and of eight bytes. The messages within CANBUS address of 0x601h outputs the data in the hexadecimal form, with each corresponding bytes, carries its own value and requires calculation as described in formulas of Section 3. The sample CANBUS messages and final processed data of the address 0x601h and 0x602h are as laid in Table 4.1 and Table 4.2.

Table 4.1: The raw CANBUS messages and final processed data of address 0x601h

Raw Message	**08**	**F4**	**2B**	**34**	**06**	**F8**	**05**	**13**
Corresponding Parameters	Motor RPM		Motor Temperature ($°C$)	Controller Temperature ($°C$)	Motor Current (A)		Battery Voltage (V)	
Processed Data	2292		43	52	178.4		129.9	

Table 4.2: The raw CANBUS messages and final processed data of address 0x602h

Raw Message	**00**	**4D**	**00**	**00**	**17**	**DB**	**00**	**00**
Corresponding Parameters	Stator Frequency (Hz)		Primary Controller Fault Code	Secondary Controller Fault Code	Throttle Input (%)	Brake Input (%)	System Bits	Not Used
Processed Data	77		00	00	23	219	00	00

As of the processed data in CAN address 0x601h and 0x602h as in Table 4.1 and also Table 4.2, the data can be utilized with direct accordance the information provided by the datasheet, and their respective values calculated directly from the formulas provided in Section 3.2.4.

4.1.2 CANBUS Data and Parameters

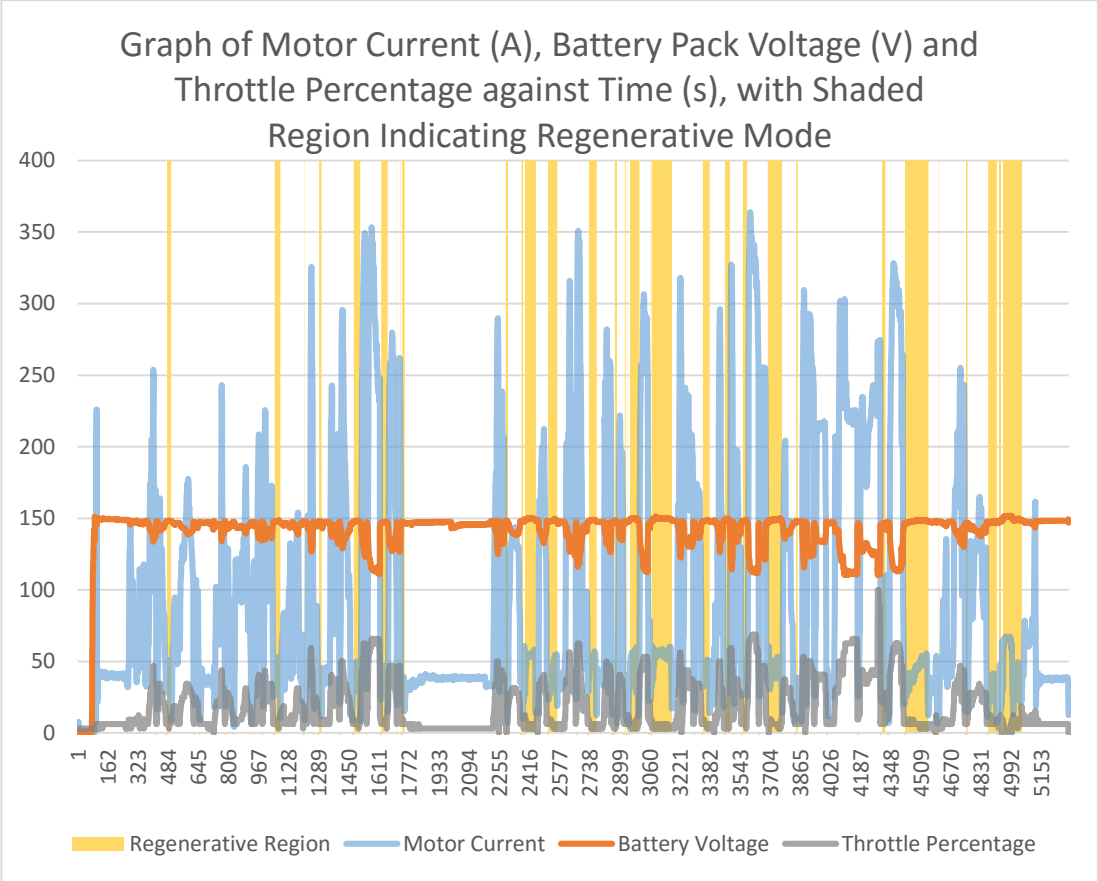


Figure 4.1: Graph of motor current, battery voltage, and battery pack percentage against time, with shaded region indicating regenerative mode

As of Figure 4.1, the overall trend shows that an increase in throttle percentage is nearly linear with the increase in motor current. This is the result of the immediate power and torque response from the motor when the user applies throttle to the vehicle. The battery pack voltage also drops in a rational relation with the increase in motor current, which denotes the electrical characteristics of an electric motor when it is being loaded. The regenerative mode also appears during the situation of having the throttle released and with the substantial motor current difference. In such situation, the motor current also drops below its usual idling current of 40 A which denotes the current is being reversed from the motor to the controller. Aside from the data shown in the graph, the bytes in byte 2 and byte 3 of CAN address 0x601h also provides vital information such as motor temperature and controller temperature respectively.

In the data obtained from CAN address 0x602h, diagnostic information such as stator frequency, a primary controller, and secondary controller fault codes can be obtained. However, it is noted that for certain parameters such as throttle input, brake input and system bits, the formula as proposed with accordance to the information from the datasheet is not applicable.

For throttle input, the CANBUS message bit idles at the hexadecimal value of 2 and peaks at a value of 22 which translates to 2 % and 34 % respectively. For the maximum throttle input of 34 %, it is suspected that the throttle calibration is set at a maximum of 34 % as on default system calibration. As the throttle input is current-sensing, a minimal amount of current is flowing even though the throttle is not depressed, which explains the minimum throttle value of 2 %. As part of throttle input fault protection, it is noted that the value of throttle input will not fall to zero. The throttle fault protection will kick in once the throttle input current falls below 0.65 mA, the fault will then be generated and any further throttle request will be zeroed for safety with adherence to EEC Regulation. Therefore, the effective throttle input for the electric vehicle configuration in this project can be recalculated as of formula 4.1.

$$Throttle\ Input\ (\%) = \frac{\left((T_H \times 16^1 + T_L \times 16^0) - 2\right)}{32} \times 100 \qquad (4.1)$$

Where

T_H = 4 higher data bits of byte 4 of address 0x602h expressed in hexadecimal form

T_L = 4 lower data bits of byte 4 of address 0x602h expressed in hexadecimal form

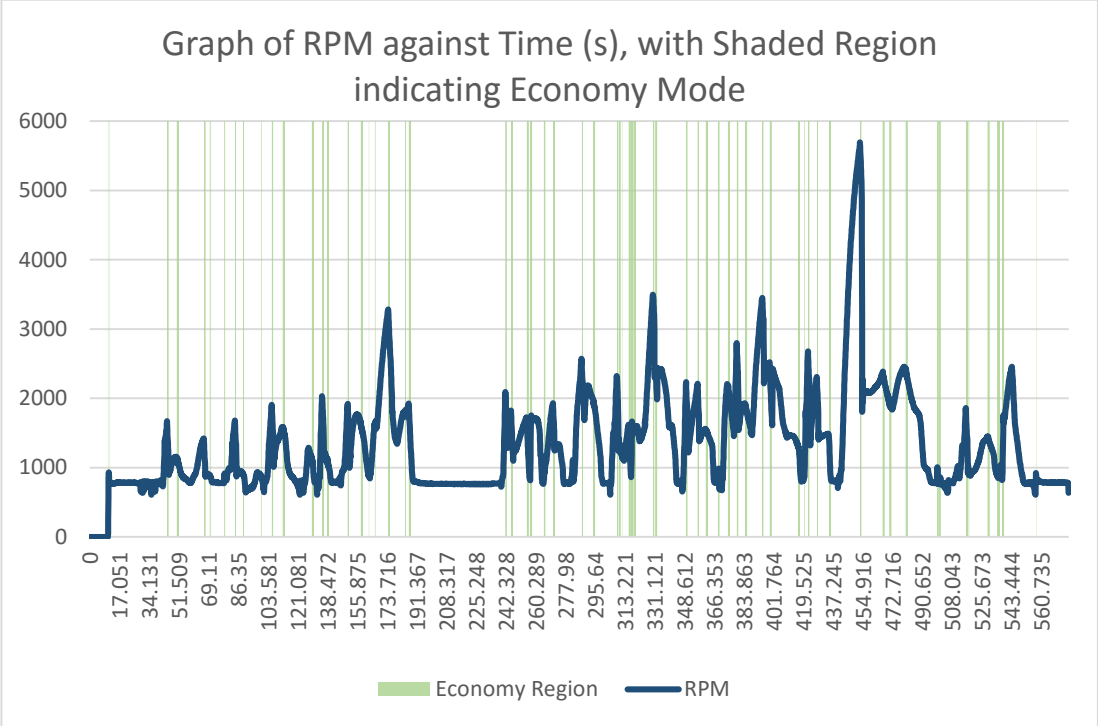


Figure 4.2: Graph of RPM against time, with shaded region indicating economy mode

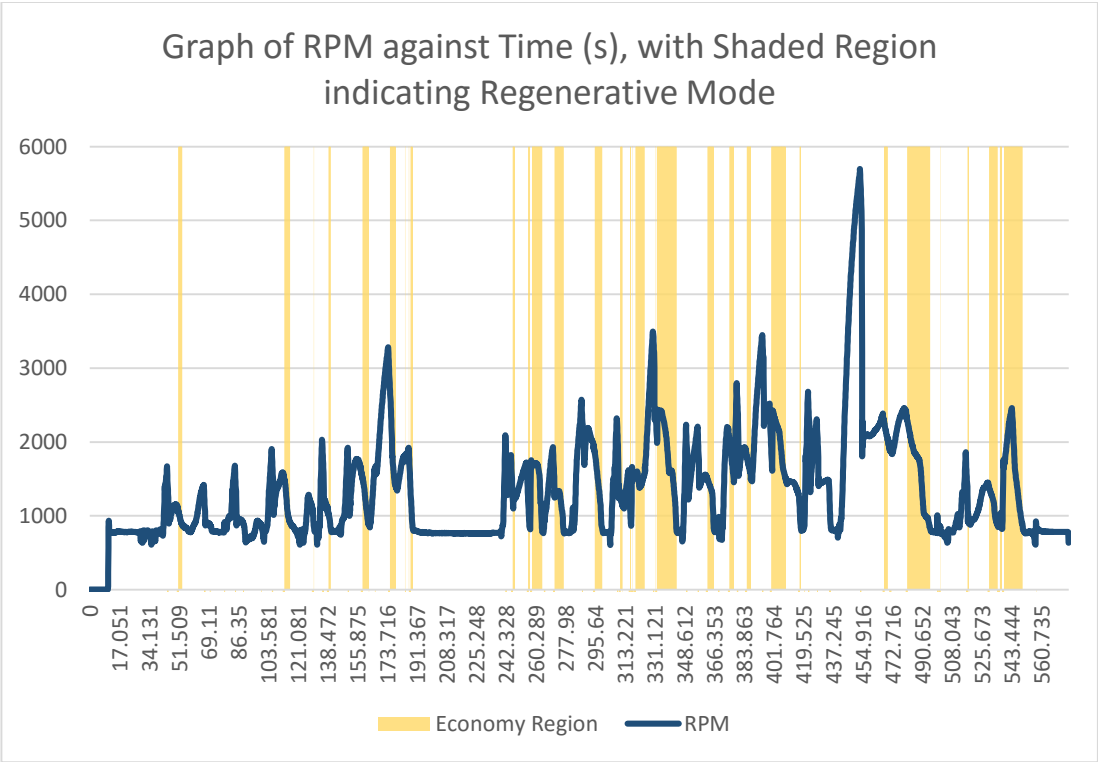


Figure 4.3: Graph of RPM against time, with shaded region indicating regenerative mode

As for the system bits, it is found that during normal operation and normal cruise, the system bits results at zero. The system normally responds by first entering into economy mode, and only with sufficient torque, the regenerative mode will only be switched to. As in Figure 4.2, letting the vehicle cruise to stop will trigger an economy bit. In this case, the system bit changes to the hexadecimal value of 0A or decimal 10 as a representation of economy driving. Once the system detected itself is out of economy state, the system bits turns zero. Besides, the regenerative mode can also be expected with harder deceleration such as going downhill, where the system bit then reflects the hexadecimal value of 2A or 42, and will switch back to bit zero once itself is out of regenerative mode. This is reflected as in Figure 4.3 where most of the regenerative mode happens during the steeper drop in of RPM region.

The data bits of byte 5 of address 0x602h which corresponds to the brake input, constantly outputs hexadecimal value of DB once the vehicle is started. The data corresponds to 219 %. It should be noted that such input should be ignored as the brake input is left disconnected during the conversion of the electric vehicle.

Time	Interface	Address	Bits	Motor RPM	Motor Temp	Controller Temperature	RMS Current	Battery Voltage
2015-09-1(	slcan0	601	[8]	808	37	46	25.8	147.3
2015-09-1(	slcan0	601	[8]	797	37	46	27.4	147.3
2015-09-1(	slcan0	601	[8]	791	37	46	32.1	146.7
2015-09-1(	slcan0	601	[8]	802	37	46	45.9	146.7
2015-09-1(	slcan0	601	[8]	814	37	46	52.9	146.2
2015-09-1(	slcan0	601	[8]	827	37	46	61.6	145.7
2015-09-1(	slcan0	601	[8]	828	37	46	83.6	145
2015-09-1(	slcan0	601	[8]	845	37	46	95.4	144.5
2015-09-1(	slcan0	601	[8]	860	37	46	105.1	144.2
2015-09-1(	slcan0	601	[8]	863	37	46	113	143.7
2015-09-1(	slcan0	601	[8]	875	37	46	123.9	142.4
2015-09-1(	slcan0	601	[8]	919	37	46	158.3	140.4
2015-09-1(	slcan0	601	[8]	934	37	46	176.1	139.3
2015-09-1(	slcan0	601	[8]	965	37	46	186.5	139.1
2015-09-1(	slcan0	601	[8]	1061	37	46	188.5	138.3
2015-09-1(	slcan0	601	[8]	1155	37	46	195.5	137.3
2015-09-1(	slcan0	601	[8]	1225	37	46	202.4	136.3
2015-09-1(	slcan0	601	[8]	1268	37	46	207.4	136
2015-09-1(	slcan0	601	[8]	1335	37	46	208.7	135.7
2015-09-1(	slcan0	601	[8]	1360	37	46	187.7	138.6
2015-09-1(	slcan0	601	[8]	1365	37	46	160.4	140.6
2015-09-1(	slcan0	601	[8]	1424	37	46	132.8	141.6

Figure 4.4: Data output of address 0x601h in comma-separated values file generated by the CANBUS data acquisition

Time	Interface	Address	Bits	Stator Frequency	Primary Controller Fault Code	Secondary Controller Fault Code	Throttle Input Percentage	Brake Input Percentage	System State
2015-09-1(	slcan0	602	[8]	28	0	0	6	219	0
2015-09-1(	slcan0	602	[8]	28	0	0	7	219	0
2015-09-1(	slcan0	602	[8]	29	0	0	8	219	0
2015-09-1(	slcan0	602	[8]	28	0	0	8	219	0
2015-09-1(	slcan0	602	[8]	28	0	0	8	219	0
2015-09-1(	slcan0	602	[8]	28	0	0	8	219	0
2015-09-1(	slcan0	602	[8]	29	0	0	9	219	0
2015-09-1(	slcan0	602	[8]	30	0	0	9	219	0
2015-09-1(	slcan0	602	[8]	30	0	0	9	219	0
2015-09-1(	slcan0	602	[8]	31	0	0	9	219	0
2015-09-1(	slcan0	602	[8]	31	0	0	9	219	0
2015-09-1(	slcan0	602	[8]	31	0	0	8	219	0
2015-09-1(	slcan0	602	[8]	32	0	0	8	219	0
2015-09-1(	slcan0	602	[8]	32	0	0	9	219	0
2015-09-1(	slcan0	602	[8]	32	0	0	9	219	0
2015-09-1(	slcan0	602	[8]	32	0	0	9	219	0
2015-09-1(	slcan0	602	[8]	32	0	0	9	219	0
2015-09-1(	slcan0	602	[8]	33	0	0	8	219	0
2015-09-1(	slcan0	602	[8]	33	0	0	8	219	0
2015-09-1(	slcan0	602	[8]	33	0	0	8	219	0
2015-09-1(	slcan0	602	[8]	33	0	0	8	219	0
2015-09-1(	slcan0	602	[8]	32	0	0	8	219	0

Figure 4.5: Data output of address 0x602h in comma-separated values file generated by the CANBUS data acquisition

The electric vehicular parameters and its values obtained from CANBUS are considerably accurate as it's of direct output from the electric vehicle controller's information network, which such information is also used by the controller itself for information processing and decision making. As of Figure 4.4 and Figure 4.5, the data output is rather comprehensive. It allows several parameters to be obtained at a point of time, which eliminates the need and the time difference of used in separate polling of data that may result in inaccuracy of data, not mentioning the speed of data outputs averages at 100 ms per address. Thus, this renders the information obtained as one of the most instantaneous, accurate, yet reliable sources of data.

4.2 GPS Data

GPS module is also installed into the system to provide geographical location data and ease of logging. The basic information that can be obtained includes the latitude and longitude. Upon the fix of 3D GPS mode, further improvised data can also be obtained, such as altitude, speed, climb and track information. As indicated in Figure 4.6, the

information updates in every second. The accuracy of the data is then further verified by using Google Maps.

Time	Latitude	Longitude	Altitude	Speed	Climb	Track	Mode
2016-01-08_16:33:25	3.03778	101.7882	69.7	0.015	0	0	3
2016-01-08_16:33:26	3.03778	101.7882	69.7	0.031	0	0	3
2016-01-08_16:33:28	3.03778	101.7882	69.7	0.185	0	0	3
2016-01-08_16:33:29	3.037782	101.7882	69.7	0.072	0	0	3
2016-01-08_16:33:30	3.037783	101.7882	69.7	0.077	0	0	3
2016-01-08_16:33:31	3.037783	101.7882	69.8	0.062	0.1	353.76	3
2016-01-08_16:33:32	3.037783	101.7882	69.8	0.057	0	351.86	3
2016-01-08_16:33:33	3.037785	101.7882	69.8	0.051	0	351.86	3
2016-01-08_16:33:34	3.037785	101.7882	69.5	0.046	-0.3	351.86	3
2016-01-08_16:33:35	3.037788	101.7882	69.3	0.031	0	351.86	3
2016-01-08_16:33:36	3.03779	101.7882	69.3	0.031	0	351.86	3
2016-01-08_16:33:37	3.037792	101.7882	69.3	0.031	0	351.86	3
2016-01-08_16:33:38	3.037792	101.7882	69.3	0.036	0	351.86	3
2016-01-08_16:33:39	3.037792	101.7882	69.3	0.041	0	351.86	3
2016-01-08_16:33:40	3.037792	101.7882	69.3	0.031	0	351.86	3
2016-01-08_16:33:42	3.037792	101.7882	69.3	0.021	0	351.86	3
2016-01-08_16:33:43	3.037795	101.7882	69.4	0.026	0.1	351.86	3
2016-01-08_16:33:44	3.037798	101.7882	69.4	0.036	0	351.86	3
2016-01-08_16:33:45	3.037803	101.7882	69.5	0.046	0.1	351.86	3
2016-01-08_16:33:46	3.037807	101.7882	69.6	0.057	0.1	351.86	3
2016-01-08_16:33:47	3.037807	101.7882	69.6	0.077	0	351.86	3
2016-01-08_16:33:48	3.037808	101.7882	69.6	0.098	0	351.86	3

Figure 4.6: Data output GPS data in comma-separated values file

Knowing the location of the vehicle, it will be convenient for the user to find out their location easily during cruising or for emergency purposes. The logging of data would also allow one to easily identify the time, vehicle location, terrain and GPS speed to be compared with other vehicle parameters such as CANBUS data to optimize the performance of the electric vehicle or for vehicle diagnostic purposes.

4.3 Individual Battery Pack Voltage and Temperature

With the 12 batteries installed in the electric vehicle, each of their voltages and temperatures is obtained and logged using the DS2436 sensors network. With such data, the driver can easily monitor the batteries individually to track down battery performance during load and even during charging. The logging of data can also be used for diagnostic purposes to track down the behaviour of the battery under various driving conditions. The time used for reading the sensors parameters is approximate a minute to obtain all the 12 sensors temperature and voltage. The data is as in Figure 4.7 and Figure 4.8.

Time	Sensor1	Sensor2	Sensor3	Sensor4	Sensor5	Sensor6	Sensor7	Sensor8	Sensor9	Sensor10	Sensor11	Sensor12
2016-02-28_19:18:38	31.47151	30.89734	31.27611	30.73798	31.1279	31.38426	30.81939	31.38323	30.67135	31.09879	31.03432	31.40048
2016-02-28_19:19:27	31.34467	31.53042	31.1696	31.26929	31.47228	30.71952	30.69362	30.61366	30.95944	31.30457	31.0052	30.91401
2016-02-28_19:20:14	31.26415	30.54511	31.11166	30.9673	31.23294	31.3539	30.76982	30.69488	31.25978	31.15797	31.13742	30.67009
2016-02-28_19:21:01	30.84793	30.96866	30.69921	31.22983	31.4933	30.5502	31.21214	30.57656	30.87971	30.83116	31.40067	31.14466
2016-02-28_19:21:46	31.35232	30.6012	31.25219	30.9709	30.68154	31.14657	31.23569	31.16792	31.02374	30.75631	31.13858	30.97197
2016-02-28_19:22:57	31.29202	30.93339	31.46535	31.21212	31.51044	30.69255	31.05993	31.24809	31.16743	30.93861	31.26955	30.71616
2016-02-28_19:24:01	30.5389	31.36732	31.14104	31.5161	30.55717	31.43535	31.16146	30.54997	30.8271	30.94141	31.00044	31.52663
2016-02-28_19:24:52	30.57386	30.85033	30.98184	31.36528	30.90881	31.27758	31.42612	30.93417	31.12844	30.71165	31.51777	30.81425
2016-02-28_19:25:59	31.20095	30.55073	31.47631	31.44052	30.74128	31.27111	31.26168	31.26919	31.42991	30.55101	30.56875	31.32793
2016-02-28_19:27:07	30.62848	30.90446	31.38666	30.8669	31.47074	31.47695	31.32022	31.42728	31.35849	31.00715	31.01643	30.89101

Figure 4.7: Data output individual battery temperature data in comma-separated values file

Time	Sensor1	Sensor2	Sensor3	Sensor4	Sensor5	Sensor6	Sensor7	Sensor8	Sensor9	Sensor10	Sensor11	Sensor12
2016-02-28_19:18:38	12.601	12.69872	13.53865	13.66576	13.83316	12.88609	12.68499	11.90909	13.38374	12.54549	12.38419	12.53974
2016-02-28_19:19:27	12.601	12.69872	13.53865	13.66576	13.83316	12.88609	13.68499	11.90909	13.38374	12.54549	12.38419	12.53974
2016-02-28_19:20:14	12.601	12.69872	13.53865	13.66576	13.83316	12.88609	14.68499	11.90909	13.38374	12.54549	12.38419	12.53974
2016-02-28_19:21:01	12.601	12.69872	13.53865	13.66576	13.83316	12.88609	15.68499	11.90909	13.38374	12.54549	12.38419	12.53974
2016-02-28_19:21:46	12.601	12.69872	13.53865	13.66576	13.83316	12.88609	16.68499	11.90909	13.38374	12.54549	12.38419	12.53974
2016-02-28_19:22:57	12.601	12.69872	13.53865	13.66576	13.83316	12.88609	17.68499	11.90909	13.38374	12.54549	12.38419	12.53974
2016-02-28_19:24:01	12.601	12.69872	13.53865	13.66576	13.83316	12.88609	18.68499	11.90909	13.38374	12.54549	12.38419	12.53974
2016-02-28_19:24:52	12.601	12.69872	13.53865	13.66576	13.83316	12.88609	19.68499	11.90909	13.38374	12.54549	12.38419	12.53974
2016-02-28_19:25:59	12.601	12.69872	13.53865	13.66576	13.83316	12.88609	20.68499	11.90909	13.38374	12.54549	12.38419	12.53974
2016-02-28_19:27:07	12.601	12.69872	13.53865	13.66576	13.83316	12.88609	21.68499	11.90909	13.38374	12.54549	12.38419	12.53974

Figure 4.8: Data output individual battery voltage data in comma-separated values file

4.4 Battery Pack Current Data

A battery pack current sensor consists of hall effect sensor is also proposed to be installed. With the availability of this sensor, the battery pack current can be obtained, which allows one to monitor the total loading on the battery pack during cruise, regenerative braking and even charging. However, due to the time constraint, the sensor is not installed onto the electric vehicle. A separate setup has been performed by using a potentiometer in place of the current sensor output to simulate the performance of the current sensor. The obtained results are deemed rational as in Figure 4.9.

Time	Battery Pack Current
2016-02-29_18:12:51	5.274542
2016-02-29_18:12:51	5.815167
2016-02-29_18:12:52	5.882163
2016-02-29_18:12:53	5.815167
2016-02-29_18:12:53	10.94169
2016-02-29_18:12:54	100.6362
2016-02-29_18:12:55	130.8152
2016-02-29_18:12:56	199.816
2016-02-29_18:12:56	226.8158
2016-02-29_18:12:57	110.8152
2016-02-29_18:12:58	72.81517
2016-02-29_18:12:58	5.815168
2016-02-29_18:13:00	4.815167
2016-02-29_18:13:01	3.815932
2016-02-29_18:13:02	5.805067
2016-02-29_18:13:02	5.825234
2016-02-29_18:13:03	5.895828
2016-02-29_18:13:04	4.815174
2016-02-29_18:13:05	4.81278
2016-02-29_18:13:05	4.816429
2016-02-29_18:13:06	4.862954
2016-02-29_18:13:07	4.889245

Figure 4.9: Comma-separated values file showing the data output from the hall-effect sensor for the parameter of battery pack current

4.5.1 Hardware Implementation onto Electric Vehicle

The battery temperature and voltage circuit, GPS module and CANBUS data acquisition board are implemented into the electric vehicle. The implementation of sensors networks is shown in Figure 4.10.

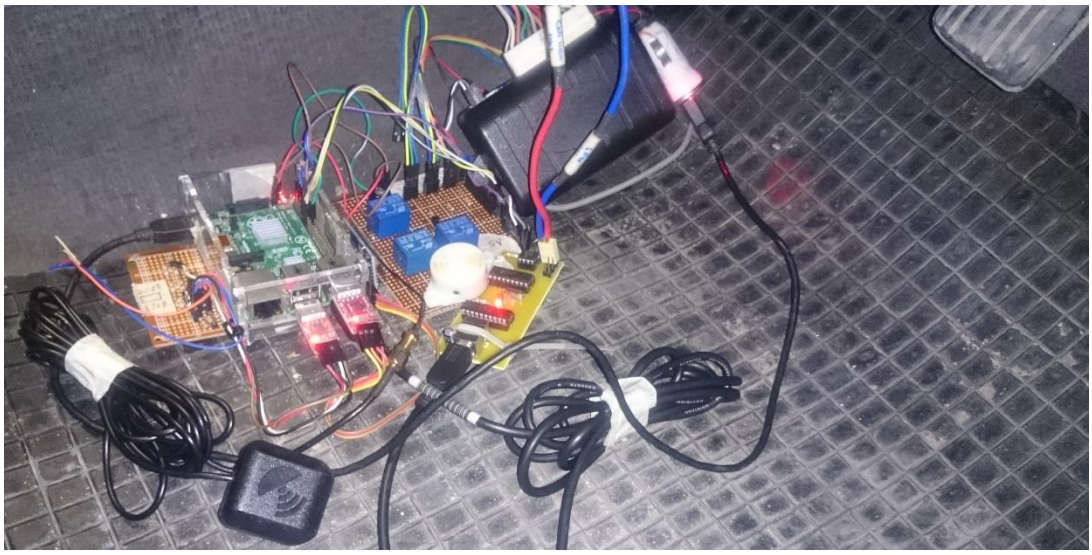


Figure 4.10: The CANBUS data acquisition board, GPS module, Dallas One-Wire bus master, power circuits and Raspberry Pi installed.

During the implementation into the electric vehicle, it is realized that the issue of battery drain is present from the individual battery sensor boards, where the circuit will drain the accessory battery continuously by the sensor circuit components such as the op-amps, optocoupler and DS 2436 voltage and temperature sensor. Besides powering the sensors, this connection also provides the reference voltage from the accessory battery to the battery sensor boards. Although the drain from the circuits is considered minute, such configuration is not recommended as this will negatively impact the lifespan of the accessory battery, not mentioning the possibility of draining down of accessory battery when vehicle is left parked without starting for weeks, which will finally result in the possibility of unable to start the electric vehicle.

However, this issue can be mitigated by adding a relay between the accessory battery and the supply connections of the sensors boards. This configuration allows

the sensors to be fed power only when the vehicle is started, and disallow current flow to the sensors boards once the car is off. This also eliminates the unnecessary risk of shorting of wiring in the case of the sensor wiring with voltage drops off and shorts to the vehicle body.

4.5.2 Electric Vehicular Systems Integration

In this project, the electric vehicular systems and sensors network now start up upon the driver unarm the car alarm, and properly shuts down when driver arms the alarm. This is achieved through the sensor output pin of the car alarm. Upon completion of start-up, a welcome chime will be sound to notify the driver the start-up is complete, and vice versa for shut down process.

The LED array outputs are also installed. As the name of the LED state implies, the corresponding LED will light up when the system state is reached. Orange LED will light up during economy driving and the CANBUS system bits output of 0A and vice versa for green LED with system bits of 2A during the regenerative mode. As during normal cruising, both LEDs will remain off. The setup is shown as in Figure 4.11.

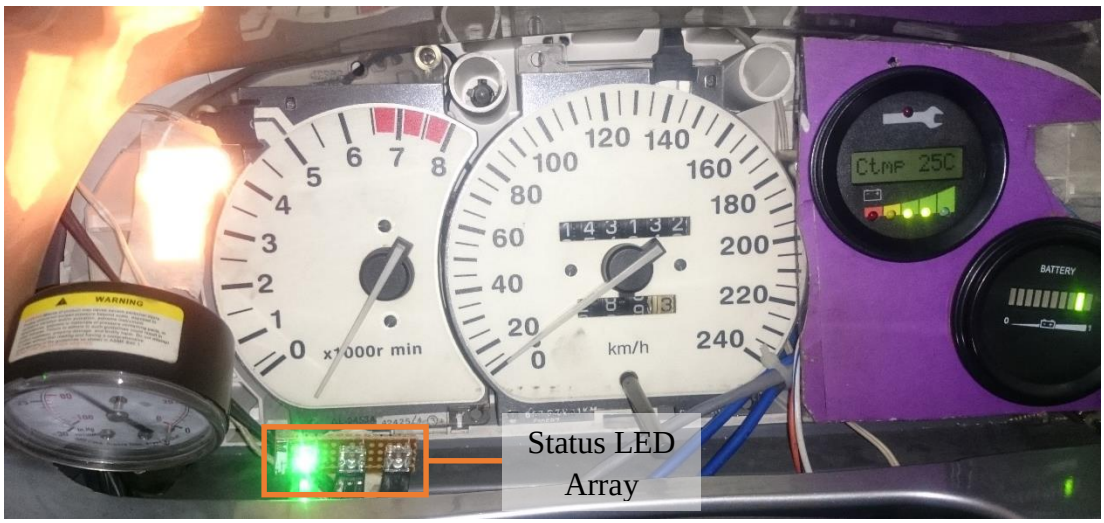


Figure 4.11: The LED array mounted on the dashboard of the electric vehicle for driver's easy viewing.

During the rise of fault associated with the electric vehicular system, the red LED will turn on and the buzzer would sound continuously as a response to alert user of faults present. The electric vehicle system, including the motor and the controller, will also be emergency shut down through cutting off the supply to the main contactor of the battery pack supply. This allows an immediate fault mitigation to prevent further delay and possibly reducing consequence system damage and personal injuries in the event of emergencies. As the fault mitigation steps accounts risks to users if executed incorrectly, yet will be dangerous if any response to fault is not performed, the fault detection system must of highly robust, reliable and quick. Therefore, the fault detection is determined through the use of CANBUS fault bits output from electric vehicle controller. Since the situation is arbitrary, the CANBUS messages simulator of USBTinViewer is used in conjunction with another CANBUS board to simulate such emergencies.

As for the log files of the sensors network, the files will be able to be accessed via any personal computer running Linux Distribution once the Raspberry Pi system is shut off. The Micro SD card can be retrieved and plugged into the computer. The CSV log files are all placed in a folder with self-explanatory file names on the logging of different sensors. For better control and simpler data analysis, the driver or technician can open the CSV log files using any spreadsheet applications such as LibreOffice Calc or Microsoft Excel to open and review the data logged.

Table 4.3: CSV Log file's name and its corresponding logged sensors

Log File Name	Logic
dataTemp	DS2436 sensors on individual battery pack
dataVolt	DS2436 sensors on individual battery pack
dataGPS	GPS module
dataCurrent	LEM Hall effect current sensor
dataCAN1	CANBUS data acquisition board
dataCAN2	CANBUS data acquisition board

CHAPTER 5

ACHIEVEMENT

5.1 Competition Participation

The author, working on hardware on the project of the electric vehicle monitoring system has collaborated with his close counterparts which are working on the graphical user interface, have participated in the inaugural Final Year Project Poster Competition 2016. The poster is shortlisted as one of the finalists in the category participated is Category of Applied Sciences.

CHAPTER 6

CONCLUSION AND RECOMMENDATIONS

6.1 Conclusion

To summarize, the author had managed to integrate the existing electric vehicular sensors network and the new CANBUS data acquisition system into place. Individual battery pack temperature and voltage measurement is achieved by using the Dallas One-Wire Sensors and its bus master. Battery pack current is also able to be measured. Location of the vehicle is also able to be determined. CANBUS data acquisition also opens up whole lot more parameters of the electric vehicle to be monitored. These includes motor RPM, motor temperature, controller temperature, motor current, battery pack voltage, motor stator frequency, controller fault codes, throttle input and system state bits. With the aid of Python programming language, the raw information from the sensors are being processed before logged into different CSV files. Drivers and technicians will benefit from the log files as it meticulously reports the parameters of the electric vehicle. The obtained data can also be cross-checked with timestamps to analyse the electric vehicle performance or provide useful information during troubleshooting of the electric vehicle.

Designed with driver experience upheld, the sensors network is now placed in close integration with the electric vehicle ignition system. The driver will now experience the ultimate convenience as the systems will be ready right before driving, providing alerts to drivers for any abnormalities and even auto shuts down properly during emergencies, all without driver's intervention as of conventional car. Thus

allow the driver to stay focused on the road and enhancing the driving experience of the electric vehicle.

Nevertheless, valuable knowledge and experience are obtained throughout the involvement in the project. These includes an in-depth understanding of vehicle wiring principles, electric vehicle hardware and systems, sensors interfacing, protocols used and the programming language to mention a few.

6.2 Recommendations

The project's main interfacing systems are developed based on Raspberry Pi Model B+, which it is used as for sensor interfacing, data acquisition, data processing, and output processing. Thus, at times of interfacing and processing, the CPU load and RAM can be very high with the execution of several scripts and with interfacing protocol loaded up to max. Not to mention on top of that, a graphic user interface also needs to be loaded in actual practice. Therefore, a faster system would be preferred to allow quicker data acquisition, especially the ones that are dealing with manual bit by bit serial execution.

Hence, the author would recommend Raspberry Pi 3, which is just released months ago (Upton, 2016). The advantage of quad-core 64-bit CPU allows faster processing and multi-threaded operations, yet leaving an identical size and price footprint, forms a perfect upgrade solution for tackling the performance issues faced in his project. Also comes with built-in Wi-Fi and Bluetooth chipset, connections to the Raspberry Pi as a secondary system is also possible, which are able accommodate passenger's entertainment and usage natively.

REFERENCES

Adrian, M. (2009). Technical: Looking inside the 1.4 12v Throttle Potentiometer / Position Sensor - The FIAT Forum. [online] FIAT Forum. Available at: http://www.fiatforum.com/bravo-brava/204546-looking-inside-1-4-12v-throttle-potentiometer-position-sensor.html [Accessed 12 Aug. 2015].

AlarmTek, J. (2012). *Car Alarms: prime security wiring diagram, shock sensor, pin switch*. *En.allexperts.com*. [online]. Available at: http://en.allexperts.com/q/Car-Alarms-1513/2012/1/prime-security-wiring-diagram.htm [Accessed 4 Mar. 2016].

Curtis, (2014). *Model 1353 CANopen Expansion Module*. 2nd ed. [ebook] New York, pp.16 - 40. Available at: http://curtisinstruments.com/Uploads/DataSheets/1353%20%2814C%291.pdf [Accessed 6 Aug. 2015]. Bakshi, A., Patnaik, P.R. and Gupta, J.K., 1992a. Pullulanase and α-amylase production by a *Bacillus cereus* isolate. *Letters in Applied Microbiology*, 14, pp. 210 – 213.

Curtis, (2015). *Standard Operating Procedures*. 1st ed. [ebook] Ontario, pp.20-65. Available at: http://evwest.com/support/Program%20Instructions%20REV%20A%20VER%205.14%20and%20up%203-27-14%20.pdf [Accessed 13 Aug. 2015].

Drafts, B. (2004). *Methods of Current Measurement*. 1st ed. Milwaukie, pp.1 - 2.

HPEVS, (2014). *CURTIS Instruments Troubleshooting Codes*. 1st ed. [ebook] Ontario, pp.3 - 12. Available at: http://www.hpevs.com/Site/images/pdf/troubleshooting/troubleshooting.pdf [Accessed 3 Sep. 2015].

Kuria, J. and Hwang, P. (2011). Investigation of Thermal Performance of Electric Vehicle BLDC Motor. *International Journal of Mechanical Engineering*, 1(1), p.14.

Lepkowski, J. (2003). *Motor Control Sensor Feedback Circuits*. 1st ed. [ebook] U.S.A., pp.1 - 9. Available at: http://ww1.microchip.com/downloads/en/AppNotes/00894a.pdf [Accessed 10 Aug. 2015].

Magana, A. and Veraguas, P. (2015). *Voltage Measurement of High Voltage Batteries for Hybrid and Electric Vehicles*. US 7982427 B2.

Manciac, A., Oprean, I., Croitorescu, G. and Fratila, G. (n.d.). Influence of Battery Voltage on Hybrid Vehicles Performances. 9(104), pp.521 - 527.

Microchip, (2003). *High-Speed CAN Transceiver*. 1st ed. [ebook] U.S.A., pp.1 - 12. Available at: http://users.ece.utexas.edu/~valvano/Datasheets/MCP2551.pdf [Accessed 6 Aug. 2015].

Microchip, (2007). *Stand-Alone CAN Controller With SPI Interface*. 1st ed. [ebook] U.S.A., pp.1-12. Available at: http://ww1.microchip.com/downloads/en/DeviceDoc/21801e.pdf [Accessed 1 Aug. 2015].

Microchip, (2015). *PIC18F13K50/14K50 Data Sheet*. 1st ed. [ebook] U.S.A., pp.1 - 31. Available at: http://ww1.microchip.com/downloads/en/devicedoc/41350c.pdf [Accessed 5 Aug. 2015].

NI, (2013). The Basics of CANopen - National Instruments. [online] Available at: http://www.ni.com/white-paper/14162/en/ [Accessed 19 Aug. 2015].

Prestolite, (n.d.). *Prestolite - Leece Neville*. [online] Prestolite.com. Available at: http://www.prestolite.com/pgs_products/specs.php?item_detail_id=879&item=8ar2200l&product=ALTERNATOR [Accessed 6 Aug. 2015].

SARMA, G. and NAGARAJU, C. (2012). Automotive Engine Temperature Control Employing Apt Temperature Measurement And Control Measures. *International Journal of Engineering Research*, 2(4), pp.1425-1429.

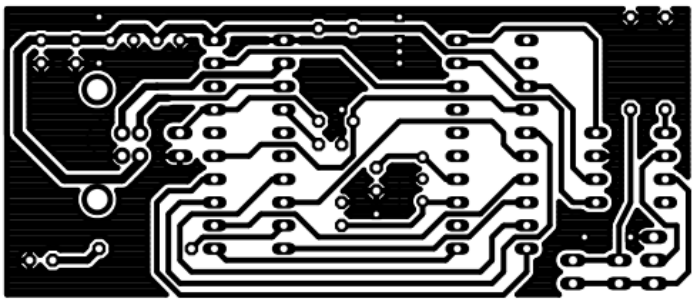
Trio Auto Accessories Sdn. Bhd.,. (2011). Retrieved from http://www.trioauto.com.my/webshaper/pcm/pictures/PowerGuard/1wayHC_1.jpg

Stevens, T. (2014). *The 2015 BMW i8 looks like an automotive artifact from the future (pictures) - Page 28 - CNET*. [online] CNET. Available at: http://www.cnet.com/pictures/the-2015-bmw-i8-looks-like-an-artifact-from-the-future-pictures/28/ [Accessed 6 Aug. 2015].

Upton, E. (2016, February 29). *Raspberry Pi 3 on sale now at $35 - Raspberry Pi.* [online]. Available at https://www.raspberrypi.org/blog/raspberry-pi-3-on-sale/ [Accessed 20 Mar. 2016].

APPENDICES

APPENDIX A: Circuit layout of CANBUS Data Acquisition Board



APPENDIX B: Python Programming Code

```
# Configuration file to check for communication with One-Wire Bus Master
# Configuration.py

import serial
import time

# Serial Connection Settings
ser = serial.Serial(
port='/dev/ttyUSB0',
baudrate=9600,
parity=serial.PARITY_NONE,
stopbits=serial.STOPBITS_ONE,
bytesize=serial.EIGHTBITS,
timeout=0.01,
)

```

```
ser.isOpen()
print "Connecting through port %s" % ser.name         # Print which port is opened

# DS2480B PRESENCE DETECTION SETTING
# Setting based on page 5 of application note DS2480B

ser.sendBreak(0.02)     # Delay 2ms
ser.flushInput()

ser.write('C1'.decode("hex"))     # Write timing byte C1
ser.sendBreak(0.02)     # Delay 2ms
ser.flushInput()

ser.write('17'.decode("hex"))     # Set PDSRC 17, Receive 16
r1=ser.read()
#r1=r1.strip()
#print r1.encode("hex")

ser.write('45'.decode("hex"))     # Set WILD 45, Receive 44
r2=ser.read()
#r2=r2.strip()
#print r2.encode("hex")

ser.write('5B'.decode("hex"))     # Set DS0/W0RT 5B, Receive 5A
r3=ser.read()
#r3=r3.strip()
#print r3.encode("hex")

ser.write('0F'.decode("hex"))     # Set RBR 0F, Receive 00
r4=ser.read()
#r4=r4.strip()
#print r4.encode("hex")

ser.write('91'.decode("hex"))     # Set OWBitResult 91, Receive 93
r5=ser.read()
#r5=r5.strip()
#print r5.encode("hex")

OneWireBitResult='93'.decode("hex") # Check for DS2480B Correct Configuration
r6=(r5==OneWireBitResult)
if r6==True:
    print "DS2480B is present!\n"
else:
    print "ERROR! DS2480B not detected!\n"

#DS2480B Presence detection settings ended here

ser.close()
print "Port %s is now closed" % ser.name         # Print which port is closed
```

```
#Main coding to obtain battery temperature and voltage
#LoopingForTesting.py
import TotalTempInArray
import TotalVoltInArray
import time

cycleValue = 1
while cycleValue < 5800:
    TotalTempInArray.TotalTempInArray()
    TotalVoltInArray.TotalVoltInArray()
    print "Cycle %s is done \n\n\n" %cycleValue
    cycleValue = cycleValue + 1
    #time.sleep(0.2)
```

```
#Function TotalTempInArray.py
import serial
import time
import temperatureValue
import recordT

RomID1=([0x1B,0x12,0x4B,0x42,0x00,0x00,0x00,0x06])
RomID2=([0x1B,0x26,0x00,0x44,0x00,0x00,0x00,0xA7])
RomID3=([0x1B,0xBF,0x66,0x10,0x00,0x00,0x00,0xDB])
RomID4=([0x1B,0xA9,0x2C,0x41,0x00,0x00,0x00,0x31])
RomID5=([0x1B,0x8C,0x26,0x4B,0x00,0x00,0x00,0xCA])
RomID6=([0x1B,0xFF,0xB8,0x2B,0x00,0x00,0x00,0x8E])
RomID7=([0x1B,0x50,0x81,0x26,0x00,0x00,0x00,0x1C])
RomID8=([0x1B,0x07,0x84,0x13,0x00,0x00,0x00,0xAC])
RomID9=([0x1B,0x2A,0x18,0x15,0x00,0x00,0x00,0xCF])
RomID10=([0x1B,0xD0,0x94,0x2C,0x00,0x00,0x00,0x43])
RomID11=([0x1B,0x0A,0x34,0x08,0x00,0x00,0x00,0x05])
RomID12=([0x1B,0x3C,0x54,0x12,0x00,0x00,0x00,0x68])

slavesNum = 1

def TotalTempInArray():
    global RomID1   #Initialize all available DS2436 Sensors a
  nd its' addresses
    global RomID2
    global RomID3
    global RomID4
    global RomID5
    global RomID6
    global RomID7
    global RomID8
    global RomID9
    global RomID10
    global RomID11
    global RomID12
    global slavesNum
```

```
36.     global tempSlave1
37.     global tempSlave2
38.     global tempSlave3
39.     global tempSlave4
40.     global tempSlave5
41.     global tempSlave6
42.     global tempSlave4
43.     global tempSlave5
44.     global tempSlave6
45.     global tempSlave7
46.     global tempSlave8
47.     global tempSlave9
48.     global tempSlave10
49.     global tempSlave11
50.     global tempSlave12
51.
52.
53.     while (slavesNum < 14):
54.         if slavesNum == 1:
55.             RomID=RomID1
56.             break
57.         elif slavesNum == 2:
58.             RomID=RomID2
59.             break
60.         elif slavesNum == 3:
61.             RomID=RomID3
62.             break
63.         elif slavesNum == 4:
64.             RomID=RomID4
65.             break
66.         elif slavesNum == 5:
67.             RomID=RomID5
68.             break
69.         elif slavesNum == 6:
70.             RomID=RomID6
71.             break
72.         elif slavesNum == 7:
73.             RomID=RomID7
74.             break
75.         elif slavesNum == 8:
76.             RomID=RomID8
77.             break
78.         elif slavesNum == 9:
79.             RomID=RomID9
80.             break
81.         elif slavesNum == 10:
82.             RomID=RomID10
83.             break
84.         elif slavesNum == 11:
85.             RomID=RomID11
86.             break
87.         elif slavesNum == 12:
88.             RomID=RomID12
89.             break
```

```
90.         elif slavesNum == 13:
91.             recordT.recordT(tempSlave1, tempSlave2, tempSlave3
   , tempSlave4, tempSlave5, tempSlave6, tempSlave7, tempSlave8,
   tempSlave9, tempSlave10, tempSlave11, tempSlave12)
92.             print "RECORDED, RETURNING TO RESET SLAVESNUM"
93.             slavesNum = 1
94.             return
95.             break
96.
97.     #print "The slave number is %d \n" % (slavesNum)
98.     temperatureVal = temperatureValue.temperatureValue(RomID)

99.         #i = 0
100.              #while i < 5:
101.              #    if (temperatureVal > 60 or temperatureVal <
    15):
102.              #        temperatureVal
103.              #        temperatureVal=temperatureValue(RomID)

104.              #        time.sleep(0.01)
105.              #    break
106.              #    i = i + 1
107.
108.              #TempArray(1,SlavesNum) = temperatureVal
109.          #print "The temperature of sensor %d is = %s  \n" %
   (slavesNum, temperatureVal)
110.
111.          while (slavesNum < 13):
112.              if slavesNum == 1:
113.                  tempSlave1 = temperatureVal
114.                  #print "Saving as %s \n" % tempSlave1
115.                  break
116.              elif slavesNum == 2:
117.                  tempSlave2 = temperatureVal
118.                  #print "Saving as %s \n" % tempSlave2
119.                  break
120.              elif slavesNum == 3:
121.                  tempSlave3 = temperatureVal
122.                  #print "Saving as %s \n" % tempSlave3
123.                  break
124.              elif slavesNum == 4:
125.                  tempSlave4 = temperatureVal
126.                  #print "Saving as %s \n" % tempSlave4
127.                  break
128.              elif slavesNum == 5:
129.                  tempSlave5 = temperatureVal
130.                  #print "Saving as %s \n" % tempSlave5
131.                  break
132.              elif slavesNum == 6:
133.                  tempSlave6 = temperatureVal
134.                  #print "Saving as %s \n" % tempSlave6
135.                  break
136.              elif slavesNum == 7:
137.                  tempSlave7 = temperatureVal
```

```
                    #print "Saving as %s \n" % tempSlave7
                    break
                elif slavesNum == 8:
                    tempSlave8 = temperatureVal
                    #print "Saving as %s \n" % tempSlave8
                    break
                elif slavesNum == 9:
                    tempSlave9 = temperatureVal
                    #print "Saving as %s \n" % tempSlave9
                    break
                elif slavesNum == 10:
                    tempSlave10 = temperatureVal
                    #print "Saving as %s \n" % tempSlave10
                    break
                elif slavesNum == 11:
                    tempSlave11 = temperatureVal
                    #print "Saving as %s \n" % tempSlave11
                    break
                elif slavesNum == 12:
                    tempSlave12 = temperatureVal
                    #print "Saving as %s \n" % tempSlave12
                    break

            slavesNum = slavesNum + 1


        #a=TempArray(1,1)
        #b=TempArray(1,2)
        #c=TempArray(1,3)
        #d=TempArray(1,4)
        #e=TempArray(1,5)
        #f=TempArray(1,6)
        #g=TempArray(1,7)
        #h=TempArray(1,8)
        #i=TempArray(1,9)
        #j=TempArray(1,10)
        #k=TempArray(1,11)
        #l=TempArray(1,12)

        #print a,b,c,d,e,f,g,h,i,j,k,l

        #return TempArray
```

```
#temperatureValue.py
import convertT
import readT
import calcTemp

def temperatureValue(RomID):
    convertT.convertT(RomID) #Request sensor to do temperature
  conversion
```

```
    [lsbTemp,msbTemp] = readT.readT(RomID)  #Read temperature from register
    finalTemperature = calcTemp.calcTemp(lsbTemp,msbTemp)  #Convert Temperature to a readable value (Check DS2436 datasheet page 11)
    return finalTemperature
```

```
# Function FlushInputBuffer.py
def FlushInputBuffer():
    import serial
    import time

    # Serial Connection Settings
    ser = serial.Serial(
    port='/dev/ttyAMA0',
    baudrate=9600,
          parity=serial.PARITY_NONE,
          stopbits=serial.STOPBITS_ONE,
          bytesize=serial.EIGHTBITS,
          timeout=0.01,
          )

          ser.isOpen()

          NumberOfFlush=0

          if NumberOfFlush<3:
              r1=ser.read()
              r1=r1.strip()
              print r1.encode("hex")
              NumberOfFlush = NumberOfFlush + 1
```

```
# Function ConvertT.py
def convertT(RomID):
    import FlushInputBuffer
    import serial
    import time

    # Serial Connection Settings
    ser = serial.Serial(
    port='/dev/ttyUSB0',
    baudrate=9600,
    parity=serial.PARITY_NONE,
    stopbits=serial.STOPBITS_ONE,
    bytesize=serial.EIGHTBITS,
    timeout=0.01,
    )
    #print "You are now in ConvertT"
    ser.isOpen()

    Result=ErrorChecking()

```

```
21.     ErrorCounter=0
22.
23.     if Result==0 and ErrorCounter<5:
24.         ErrorChecking()
25.         ErrorCounter=ErrorCounter+1
26.
27.         if ErrorCounter==4:
28.             print "Error in communicating with slaves\n"
29.
30.     ser.write('C1'.decode("hex"))     # Set RESET C1, Receive
   CD
31.     r1=ser.read()
32.     r1=r1.strip()
33.     #print r1.encode("hex")
34.
35.     ser.write('E1'.decode("hex"))     # Set DATA MODE, Receive
    NOTHING
36.     r2=ser.read()
37.     r2=r2.strip()
38.     #print r2.encode("hex")
39.
40.     ser.write('55'.decode("hex"))     # Set SKIP ROM 55, Recei
   ve 55
41.     r3=ser.read()
42.     r3=r3.strip()
43.     #print r3.encode("hex")
44.
45.     ser.write((RomID))    # Write RomID, Receive RomID
46.
47.     i = 1
48.     while i < 9:
49.         r4=ser.read()
50.         r4=r4.strip()
51.         #print "The value i = %s, current r4 = %r  \n" % (i, r
   4.encode("hex"))
52.         i = i + 1
53.
54.     ser.write('E3'.decode("hex"))     # Set COMAND MODE E3, Re
   ceive NOTHING
55.     r5=ser.read()
56.     r5=r5.strip()
57.     #print "The commandmode E3, Receive Nothing %s:  \n" % r5.
   encode("hex")
58.
59.     ser.write('EF'.decode("hex"))     # Set ARM STRONG PULLUP
   EF, Receive NOTHING
60.     r6=ser.read()
61.     r6=r6.strip()
62.     #print "The ARMPULLUP EF, Receive Nothing %s:  \n" % r6.en
   code("hex")
63.
64.     ser.write('F1'.decode("hex"))     # Set TERMINATE PULSE F1
   , Receive EF
65.     r7=ser.read()
```

```
66.     r7=r7.strip()
67.     #print "The TerminatePulse F1, Receive EF %s:  \n" % r7.en
   code("hex")
68.
69.     ser.write('E1'.decode("hex"))     # Set DATA MODE E1, Rece
   ive NOTHING
70.     r8=ser.read()
71.     r8=r8.strip()
72.     #print "The data mode E1, Receive Nothing %s:  \n" % r8.en
   code("hex")
73.
74.     ser.write('D2'.decode("hex"))     # Set CONVERT TEMPERATUR
   E, Receive D2
75.     time.sleep(0.015)                 # Conversion require at
   least 10ms
76.     r9=ser.read()
77.     r9=r9.strip()
78.     #print "The converttemperature D2, Receive D2 %s:  \n" % r
   9.encode("hex")
79.
80.     r10=ser.read()                    # POLL FOR RESPONSE PULS
   E, Receive F6
81.     r10=r10.strip()
82.     #print "Receive RESPONSEPULSE F6 %s:  \n" % r10.encode("he
   x")
83.
84.     ser.write('E3'.decode("hex"))     # Set COMAND MODE E3, Re
   ceive NOTHING
85.     r11=ser.read()
86.     r11=r11.strip()
87.     #print "The command mode E3, Receive Nothing %s:  \n" % r1
   1.encode("hex")
88.
89.     ser.write('ED'.decode("hex"))     # Set DISARM STRONG PULL
   UP ED, Receive NOTHING
90.     r12=ser.read()
91.     r12=r6.strip()
92.     #print "The command mode ED, Receive Nothing %s:  \n" % r1
   2.encode("hex")
93.
94.     ser.write('F1'.decode("hex"))     # Set TERMINATE PULSE F1
   , Receive EF
95.     r13=ser.read()
96.     r13=r13.strip()
97.     #print "The terminatepulse F1, Receive EF %s:  \n" % r13.e
   ncode("hex")
98.
99.     ser.write('C1'.decode("hex"))     # Set RESET C1, Receive
   CD
100.          r14=ser.read()
101.          r14=r14.strip()
102.          #print "The C1, ReceiveCD %s:  \n" % r14.encode("hex
   ")
103.
```

```
104.          r15=ser.read()
105.          r15=r15.strip()
106.          #print "Receive Nothing %s:  \n" % r15.encode("hex")

107.
108.          #print "Done temperature conversion!\n"
109.
110.
111.      def ErrorChecking():
112.          import serial
113.          import FlushInputBuffer
114.
115.          # Serial Connection Settings
116.          ser = serial.Serial(
117.          port='/dev/ttyUSB0',
118.          baudrate=9600,
119.          parity=serial.PARITY_NONE,
120.          stopbits=serial.STOPBITS_ONE,
121.          bytesize=serial.EIGHTBITS,
122.          timeout=0.01,
123.          )
124.
125.          ser.isOpen()
126.
127.          FlushInputBuffer.FlushInputBuffer()
128.
129.          ser.write('E3'.decode("hex"))     # Set COMMAND MODE
    E3, before data interfacing as precaution
130.          r16 = ser.read()
131.          r16 = r16.strip()
132.          #print r16.encode("hex")
133.
134.          ser.write('33'.decode("hex"))     # Set PULLUP DURAT
   ION=524ms, Receive 32
135.          r17 = ser.read()
136.          r17 = r17.strip()
137.          #print r17.encode("hex")
138.
139.          DataShouldBeReceived='32'.decode("hex") # Check for
   Correct Data Received, PULLUP DUR=65.6ms
140.          r18 = (r17 == DataShouldBeReceived)
141.          if r18 == True:
142.              #print "Correct Data Received\n"
143.              Result = 1
144.          else:
145.              print "ERROR! Resend Command Code\n"
146.              Result = 0
```

```
1. #function readT.py
2. def readT(RomID):
3.     import FlushInputBuffer
4.     import serial
```

```
5.      import time
6.
7.      # Serial Connection Settings
8.      ser = serial.Serial(
9.      port='/dev/ttyUSB0',
10.     baudrate=9600,
11.     parity=serial.PARITY_NONE,
12.     stopbits=serial.STOPBITS_ONE,
13.     bytesize=serial.EIGHTBITS,
14.     timeout=0.01,
15.     )
16.     #print "You are now in ReadT"
17.     ser.isOpen()
18.     r0=ser.read()
19.     r0=r0.strip()
20.     #print "The serial port reopen and reads %s:  \n" % r0.enc
   ode("hex")
21.
22.     FlushInputBuffer.FlushInputBuffer()
23.
24.     ser.write('C1'.decode("hex"))     # Set RESET C1, Receive
   CD
25.     r1=ser.read()
26.     r1=r1.strip()
27.     #print "The reset C1, Receive CD %s:  \n" % r1.encode("hex
   ")
28.
29.     ser.write('E1'.decode("hex"))     # Set DATA MODE, Receive
    NOTHING
30.     r2=ser.read()
31.     r2=r2.strip()
32.     #print "The reset DATAMODE E1, Receive NOTHING %s:  \n" %
   r2.encode("hex")
33.
34.     ser.write('55'.decode("hex"))     # Set SKIP ROM 55, Recei
   ve 55
35.     r3=ser.read()
36.     r3=r3.strip()
37.     #print "Receive 55 %s:  \n" % r3.encode("hex")
38.
39.     #print "The current RomID at LoopingForTesting is = %s:  \
   n" % RomID
40.
41.     ser.write((RomID))    # Write RomID, Receive RomID
42.
43.     i = 1
44.     while i < 9:
45.         r4=ser.read()
46.         r4=r4.strip()
47.         print "The value i = %s, current r4 = %r  \n" % (i, r4
   .encode("hex"))
48.         i = i + 1
49.
```

```
50.     ser.write('B2'.decode("hex"))     # Set READ REGISTER B2,
   Receive B2
51.     r5=ser.read()
52.     r5=r5.strip()
53.     #print "The read register B2, Receive B2 %s:  \n" % r5.enc
   ode("hex")
54.
55.     ser.write('60'.decode("hex"))     # Set TEMPERATURE ADDRES
   S 60, Receive 60
56.     r6=ser.read()
57.     r6=r6.strip()
58.     #print "The TempAdddress 60, Receive 60 %s:  \n" % r6.enco
   de("hex")
59.
60.     ser.write('FF'.decode("hex"))     # Obtain LSB TEMPERATURE

61.     r7=ser.read()
62.     lsbTemp=r7.strip()
63.     #print "The lsbTemp of battery value = %s:  \n" % lsbTemp.
   encode("hex")
64.     lsbTemp = bin(int(lsbTemp.encode("hex"),16))[2:]
65.     #print lsbTemp
66.
67.     ser.write('FF'.decode("hex"))     # Obtain MSB TEMPERATURE

68.     r8=ser.read()
69.     msbTemp=r8.strip()
70.     #print "The msbTemp of battery value = %s:  \n" % msbTemp.
   encode("hex")
71.     msbTemp = int(msbTemp.encode("hex"),16)
72.     #print msbTemp
73.
74.     ser.write('E3'.decode("hex"))     # Set COMAND MODE E3, Re
   ceive NOTHING
75.     r9=ser.read()
76.     r9=r9.strip()
77.     #print "The commandmode E3, Receive NOTHING %s:  \n" % r9.
   encode("hex")
78.
79.     ser.write('C1'.decode("hex"))     # Set RESET PULSE C1, Re
   ceive CD
80.     r10=ser.read()
81.     r10=r10.strip()
82.     #print "The reset C1, Receive CD %s:  \n" % r10.encode("he
   x")
83.
84.     return lsbTemp,msbTemp
85.
86.     #print "Done temperature conversion!\n"
```

```
1. #Function calcTemp.py
2. def calcTemp(lsbTemp,msbTemp):
```

```
3.
4.      #print "You are now in calcTemp"
5.
6.      lsbTemp = bitConv(lsbTemp)
7.
8.      #print "The lsbTemp new value = %s:  \n" % lsbTemp
9.
10.     finalTemperature = float (msbTemp) + lsbTemp
11.
12.     #print "The finalTemp is = %s:  \n" % finalTemperature
13.     #print "Done temperature calculation!\n"
14.
15.     return finalTemperature
16.
17. def bitConv(b):
18.     #print "The bitConv2 initial lsbTemp value = %s:  \n" % b

19.     return int(b, 2) / 2.**(len(b))
```

```
1. #Function record.py
2. def recordT(tempSlave1, tempSlave2, tempSlave3, tempSlave4, te
   mpSlave5, tempSlave6, tempSlave7, tempSlave8, tempSlave9, temp
   Slave10, tempSlave11, tempSlave12):
3.
4.      import os
5.      import time
6.      import datetime
7.
8.      #print "You are now in recordT"
9.      file = open("/home/pi/Desktop/dataTemp.csv","a")
10.     i = 0
11.
12.     if os.stat("/home/pi/Desktop/dataTemp.csv").st_size == 0:

13.         file.write("Time,Sensor1,Sensor2,Sensor3,Sensor4,Senso
   r5,Sensor6,Sensor7,Sensor8,Sensor9,Sensor10,Sensor11,Sensor12\
   n")
14.
15.     while(i < 1):
16.         timeRec = datetime.datetime.now().strftime("%Y-%m-%d_%
   H:%M:%S")
17.         file.write(str(timeRec)+","+str(tempSlave1)+","+str(te
   mpSlave2)+","+str(tempSlave3)+","+str(tempSlave4)+","+str(temp
   Slave5)+","+str(tempSlave6)+","+str(tempSlave7)+","+str(tempSl
   ave8)+","+str(tempSlave9)+","+str(tempSlave10)+","+str(tempSla
   ve11)+","+str(tempSlave12)+"\n")
18.         i = i + 1
19.         file.close
20.     #print "Printing Done.\n"
```

```
1. #Function TotalVoltInArray.py
2. import serial
3. import time
4. import voltageValue
5. import recordV
6.
7. RomID1=([0x1B,0x12,0x4B,0x42,0x00,0x00,0x00,0x06])
8. RomID2=([0x1B,0x26,0x00,0x44,0x00,0x00,0x00,0xA7])
9. RomID3=([0x1B,0xBF,0x66,0x10,0x00,0x00,0x00,0xDB])
10. RomID4=([0x1B,0xA9,0x2C,0x41,0x00,0x00,0x00,0x31])
11. RomID5=([0x1B,0x8C,0x26,0x4B,0x00,0x00,0x00,0xCA])
12. RomID6=([0x1B,0xFF,0xB8,0x2B,0x00,0x00,0x00,0x8E])
13. RomID7=([0x1B,0x50,0x81,0x26,0x00,0x00,0x00,0x1C])
14. RomID8=([0x1B,0x07,0x84,0x13,0x00,0x00,0x00,0xAC])
15. RomID9=([0x1B,0x2A,0x18,0x15,0x00,0x00,0x00,0xCF])
16. RomID10=([0x1B,0xD0,0x94,0x2C,0x00,0x00,0x00,0x43])
17. RomID11=([0x1B,0x0A,0x34,0x08,0x00,0x00,0x00,0x05])
18. RomID12=([0x1B,0x3C,0x54,0x12,0x00,0x00,0x00,0x68])
19.
20. slavesNum = 1
21.
22. def TotalVoltInArray():
23.     global RomID1   #Initialize all available DS2436 Sensors a
    nd its' addresses
24.     global RomID2
25.     global RomID3
26.     global RomID4
27.     global RomID5
28.     global RomID6
29.     global RomID7
30.     global RomID8
31.     global RomID9
32.     global RomID10
33.     global RomID11
34.     global RomID12
35.     global slavesNum
36.     global voltSlave1
37.     global voltSlave2
38.     global voltSlave3
39.     global voltSlave4
40.     global voltSlave5
41.     global voltSlave6
42.     global voltSlave4
43.     global voltSlave5
44.     global voltSlave6
45.     global voltSlave7
46.     global voltSlave8
47.     global voltSlave9
48.     global voltSlave10
49.     global voltSlave11
50.     global voltSlave12
51.
52.
53.     while (slavesNum < 14):
```

```
54.         if slavesNum == 1:
55.             RomID=RomID1
56.             Ratio=0.5198
57.             break
58.         elif slavesNum == 2:
59.             RomID=RomID2
60.             Ratio=0.5158
61.             break
62.         elif slavesNum == 3:
63.             RomID=RomID3
64.             Ratio=0.4838
65.             break
66.         elif slavesNum == 4:
67.             RomID=RomID4
68.             Ratio=0.4793
69.             break
70.         elif slavesNum == 5:
71.             RomID=RomID5
72.             Ratio=0.4735
73.             break
74.         elif slavesNum == 6:
75.             RomID=RomID6
76.             Ratio=0.5083
77.             break
78.         elif slavesNum == 7:
79.             RomID=RomID7
80.             Ratio=0.473
81.             break
82.         elif slavesNum == 8:
83.             RomID=RomID8
84.             Ratio=0.55
85.             break
86.         elif slavesNum == 9:
87.             RomID=RomID9
88.             Ratio=0.4894
89.             break
90.         elif slavesNum == 10:
91.             RomID=RomID10
92.             Ratio=0.5221
93.             break
94.         elif slavesNum == 11:
95.             RomID=RomID11
96.             Ratio=0.5289
97.             break
98.         elif slavesNum == 12:
99.             RomID=RomID12
100.                 Ratio=0.4215
101.                 break
102.             elif slavesNum == 13:
103.                 recordV.recordV(voltSlave1, voltSlave2, volt
    Slave3, voltSlave4, voltSlave5, voltSlave6, voltSlave7, voltSl
    ave8, voltSlave9, voltSlave10, voltSlave11, voltSlave12)
104.                 print "RECORDED, RETURNING TO RESET SLAVESNU
    M"
```

```
105.                 slavesNum = 1
106.                 return
107.                 break
108.
109.         #print "The slave number is %d \n" % (slavesNum)
110.         voltageVal = voltageValue.voltageValue(RomID,Ratio)

111.             #i = 0
112.             #while i < 5:
113.             #    if (temperatureVal > 60 or temperatureVal <
    15):
114.             #        temperatureVal
115.             #        temperatureVal=temperatureValue(RomID)

116.             #        time.sleep(0.01)
117.             #    break
118.             #    i = i + 1
119.
120.             #TempArray(1,SlavesNum) = temperatureVal
121.         #print "The voltage of sensor %d is = %s  \n" % (sla
   vesNum, voltageVal)
122.
123.
124.         while (slavesNum < 13):
125.             if slavesNum == 1:
126.                 voltSlave1 = voltageVal
127.                 #print "Saving as %s \n" % voltSlave1
128.                 break
129.             elif slavesNum == 2:
130.                 voltSlave2 = voltageVal
131.                 #print "Saving as %s \n" % voltSlave2
132.                 break
133.             elif slavesNum == 3:
134.                 voltSlave3 = voltageVal
135.                 #print "Saving as %s \n" % voltSlave3
136.                 break
137.             elif slavesNum == 4:
138.                 voltSlave4 = voltageVal
139.                 #print "Saving as %s \n" % voltSlave4
140.                 break
141.             elif slavesNum == 5:
142.                 voltSlave5 = voltageVal
143.                 #print "Saving as %s \n" % voltSlave5
144.                 break
145.             elif slavesNum == 6:
146.                 voltSlave6 = voltageVal
147.                 #print "Saving as %s \n" % voltSlave6
148.                 break
149.             elif slavesNum == 7:
150.                 voltSlave7 = voltageVal
151.                 #print "Saving as %s \n" % voltSlave7
152.                 break
153.             elif slavesNum == 8:
154.                 voltSlave8 = voltageVal
```

```
155.                 #print "Saving as %s \n" % voltSlave8
156.                 break
157.             elif slavesNum == 9:
158.                 voltSlave9 = voltageVal
159.                 #print "Saving as %s \n" % voltSlave9
160.                 break
161.             elif slavesNum == 10:
162.                 voltSlave10 = voltageVal
163.                 #print "Saving as %s \n" % voltSlave10
164.                 break
165.             elif slavesNum == 11:
166.                 voltSlave11 = voltageVal
167.                 #print "Saving as %s \n" % voltSlave11
168.                 break
169.             elif slavesNum == 12:
170.                 voltSlave12 = voltageVal
171.                 #print "Saving as %s \n" % voltSlave12
172.                 break
173.
174.
175.         slavesNum = slavesNum + 1
176.
177.     #a=TempArray(1,1)
178.     #b=TempArray(1,2)
179.     #c=TempArray(1,3)
180.     #d=TempArray(1,4)
181.     #e=TempArray(1,5)
182.     #f=TempArray(1,6)
183.     #g=TempArray(1,7)
184.     #h=TempArray(1,8)
185.     #i=TempArray(1,9)
186.     #j=TempArray(1,10)
187.     #k=TempArray(1,11)
188.     #l=TempArray(1,12)
189.
190.     #print a,b,c,d,e,f,g,h,i,j,k,l
191.
192.     #return TempArray
```

```
1. #Function voltageValue.py
2. import convertV
3. import readV
4. import calcVolt
5.
6. def voltageValue(RomID,Ratio):
7.     convertV.convertV(RomID) #Request sensor to do voltage con
   version
8.     [lsbVolt,msbVolt] = readV.readV(RomID)  #Read temperature
   from register
9.     finalVolt = calcVolt.calcVolt(lsbVolt,msbVolt,Ratio)  #Con
   vert Temperature to a readable value (Check DS2436 datasheet p
   age 11)
```

```
    return finalVolt
```

```
#Function convertV.py
def convertV(RomID):
    import FlushInputBuffer
    import serial
    import time

    # Serial Connection Settings
    ser = serial.Serial(
    port='/dev/ttyUSB0',
    baudrate=9600,
    parity=serial.PARITY_NONE,
    stopbits=serial.STOPBITS_ONE,
    bytesize=serial.EIGHTBITS,
    timeout=0.01,
    )
    #print "You are now in ConvertV"
    ser.isOpen()

    Result=ErrorChecking()

    ErrorCounter=0

    if Result==0 and ErrorCounter<5:
        ErrorChecking()
        ErrorCounter=ErrorCounter+1

        if ErrorCounter==4:
            print "Error in communicating with slaves\n"

    ser.write('C1'.decode("hex"))     # Set RESET C1, Receive
CD
    r1=ser.read()
    r1=r1.strip()
    #print r1.encode("hex")

    ser.write('E1'.decode("hex"))     # Set DATA MODE, Receive
 NOTHING
    r2=ser.read()
    r2=r2.strip()
    #print r2.encode("hex")

    ser.write('55'.decode("hex"))     # Set SKIP ROM 55, Recei
ve 55
    r3=ser.read()
    r3=r3.strip()
    #print r3.encode("hex")

    ser.write((RomID))    # Write RomID, Receive RomID

    i = 1
```

```
48.     while i < 9:
49.         r4=ser.read()
50.         r4=r4.strip()
51.         #print "The value i = %s, current r4 = %r  \n" % (i, r
    4.encode("hex"))
52.         i = i + 1
53.
54.     ser.write('E3'.decode("hex"))     # Set COMAND MODE E3, Re
    ceive NOTHING
55.     r5=ser.read()
56.     r5=r5.strip()
57.     #print "The commandmode E3, Receive Nothing %s:  \n" % r5.
    encode("hex")
58.
59.     ser.write('EF'.decode("hex"))     # Set ARM STRONG PULLUP
    EF, Receive NOTHING
60.     r6=ser.read()
61.     r6=r6.strip()
62.     #print "The ARMPULLUP EF, Receive Nothing %s:  \n" % r6.en
    code("hex")
63.
64.     ser.write('F1'.decode("hex"))     # Set TERMINATE PULSE F1
    , Receive EF
65.     r7=ser.read()
66.     r7=r7.strip()
67.     #print "The TerminatePulse F1, Receive EF %s:  \n" % r7.en
    code("hex")
68.
69.     ser.write('E1'.decode("hex"))     # Set DATA MODE E1, Rece
    ive NOTHING
70.     r8=ser.read()
71.     r8=r8.strip()
72.     #print "The data mode E1, Receive Nothing %s:  \n" % r8.en
    code("hex")
73.
74.     ser.write('B4'.decode("hex"))     # Set CONVERT VOLTAGE, R
    eceive B4
75.     time.sleep(0.015)                 # Conversion require at
    least 10ms
76.     r9=ser.read()
77.     r9=r9.strip()
78.     #print "The convert Voltage B4, Receive B4 %s:  \n" % r9.e
    ncode("hex")
79.
80.     r10=ser.read()                    # POLL FOR RESPONSE PULS
    E, Receive F6
81.     r10=r10.strip()
82.     #print "Receive RESPONSEPULSE F6 %s:  \n" % r10.encode("he
    x")
83.
84.     ser.write('E3'.decode("hex"))     # Set COMAND MODE E3, Re
    ceive NOTHING
85.     r11=ser.read()
86.     r11=r11.strip()
```

```
87.     #print "The command mode E3, Receive Nothing %s:  \n" % r1
   1.encode("hex")
88.
89.     ser.write('ED'.decode("hex"))     # Set DISARM STRONG PULL
   UP ED, Receive NOTHING
90.     r12=ser.read()
91.     r12=r6.strip()
92.     #print "The command mode ED, Receive Nothing %s:  \n" % r1
   2.encode("hex")
93.
94.     ser.write('F1'.decode("hex"))     # Set TERMINATE PULSE F1
   , Receive EF
95.     r13=ser.read()
96.     r13=r13.strip()
97.     #print "The terminatepulse F1, Receive EF %s:  \n" % r13.e
   ncode("hex")
98.
99.     ser.write('C1'.decode("hex"))     # Set RESET C1, Receive
   CD
100.          r14=ser.read()
101.          r14=r14.strip()
102.          #print "The C1, ReceiveCD %s:  \n" % r14.encode("hex
   ")
103.
104.          r15=ser.read()
105.          r15=r15.strip()
106.          #print "Receive Nothing %s:  \n" % r15.encode("hex")

107.
108.          #print "Done voltage conversion!\n"
109.
110.
111.      def ErrorChecking():
112.          import serial
113.          import FlushInputBuffer
114.
115.          # Serial Connection Settings
116.          ser = serial.Serial(
117.          port='/dev/ttyUSB0',
118.          baudrate=9600,
119.          parity=serial.PARITY_NONE,
120.          stopbits=serial.STOPBITS_ONE,
121.          bytesize=serial.EIGHTBITS,
122.          timeout=0.01,
123.          )
124.
125.          ser.isOpen()
126.
127.          FlushInputBuffer.FlushInputBuffer()
128.
129.          ser.write('E3'.decode("hex"))     # Set COMMAND MODE
    E3, before data interfacing as precaution
130.          r16 = ser.read()
131.          r16 = r16.strip()
```

```
132.          #print r16.encode("hex")
133.
134.          ser.write('33'.decode("hex"))     # Set PULLUP DURAT
   ION=524ms, Receive 32
135.          r17 = ser.read()
136.          r17 = r17.strip()
137.          #print r17.encode("hex")
138.
139.          DataShouldBeReceived='32'.decode("hex") # Check for
   Correct Data Received, PULLUP DUR=65.6ms
140.          r18 = (r17 == DataShouldBeReceived)
141.          if r18 == True:
142.              #print "Correct Data Received\n"
143.              Result = 1
144.          else:
145.              print "ERROR! Resend Command Code\n"
146.              Result = 0
```

```
1. #Function readV.py
2. def readV(RomID):
3.     import FlushInputBuffer
4.     import serial
5.     import time
6.
7.     # Serial Connection Settings
8.     ser = serial.Serial(
9.     port='/dev/ttyUSB0',
10.     baudrate=9600,
11.     parity=serial.PARITY_NONE,
12.     stopbits=serial.STOPBITS_ONE,
13.     bytesize=serial.EIGHTBITS,
14.     timeout=0.01,
15.     )
16.     #print "You are now in ReadV"
17.     ser.isOpen()
18.     r0=ser.read()
19.     r0=r0.strip()
20.     #print "The serial port reopen and reads %s:  \n" % r0.enc
   ode("hex")
21.
22.     FlushInputBuffer.FlushInputBuffer()
23.
24.     ser.write('C1'.decode("hex"))     # Set RESET C1, Receive
   CD
25.     r1=ser.read()
26.     r1=r1.strip()
27.     #print "The reset C1, Receive CD %s:  \n" % r1.encode("hex
   ")
28.
29.     ser.write('E1'.decode("hex"))     # Set DATA MODE, Receive
    NOTHING
30.     r2=ser.read()
```

```
31.     r2=r2.strip()
32.     #print "The reset DATAMODE E1, Receive NOTHING %s:  \n" %
   r2.encode("hex")
33.
34.     ser.write('55'.decode("hex"))     # Set SKIP ROM 55, Recei
   ve 55
35.     r3=ser.read()
36.     r3=r3.strip()
37.     #print "Receive 55 %s:  \n" % r3.encode("hex")
38.
39.     #print "The current RomID at LoopingForTesting is = %s:  \
   n" % RomID
40.
41.     ser.write((RomID))    # Write RomID, Receive RomID
42.
43.     i = 1
44.     while i < 9:
45.         r4=ser.read()
46.         r4=r4.strip()
47.         print "The value i = %s, current r4 = %r  \n" % (i, r4
   .encode("hex"))
48.         i = i + 1
49.
50.     ser.write('B2'.decode("hex"))     # Set READ REGISTER B2,
   Receive B2
51.     r5=ser.read()
52.     r5=r5.strip()
53.     #print "The read register B2, Receive B2 %s:  \n" % r5.enc
   ode("hex")
54.
55.     ser.write('77'.decode("hex"))     # Set VOLTAGE ADDRESS 77
   , Receive 77
56.     r6=ser.read()
57.     r6=r6.strip()
58.     #print "The Voltage Address 77, Receive 77 %s:  \n" % r6.e
   ncode("hex")
59.
60.     ser.write('FF'.decode("hex"))     # Obtain LSB VOLTAGE
61.     r7=ser.read()
62.     lsbVolt=r7.strip()
63.     lsbVolt=lsbVolt.encode("hex")
64.     #print "The lsbVolt of battery value = %s:  \n" % lsbVolt

65.
66.     ser.write('FF'.decode("hex"))     # Obtain MSB VOLTAGE
67.     r8=ser.read()
68.     msbVolt=r8.strip()
69.     msbVolt=msbVolt.encode("hex")
70.     #print "The msbVolt of battery value = %s:  \n" % msbVolt

71.
72.     ser.write('E3'.decode("hex"))     # Set COMAND MODE E3, Re
   ceive NOTHING
73.     r9=ser.read()
```

```
74.     r9=r9.strip()
75.     #print "The commandmode E3, Receive NOTHING %s:  \n" % r9.
   encode("hex")
76.
77.     ser.write('C1'.decode("hex"))     # Set RESET PULSE C1, Re
   ceive CD
78.     r10=ser.read()
79.     r10=r10.strip()
80.     #print "The reset C1, Receive CD %s:  \n" % r10.encode("he
   x")
81.
82.     return lsbVolt,msbVolt
83.
84.     #print "Done voltage reading!\n"
```

```
1. #Function calcVolt.py
2. def calcVolt(lsbVolt,msbVolt,Ratio):
3.
4.     #print "You are now in calcVolt"
5.
6.     if len(lsbVolt) == 1:
7.         lsbVolt = str(0) + lsbVolt
8.
9.     combinedVolt = msbVolt + lsbVolt
10.     #print "The combinedVolt added together = %s:  \n" % combi
   nedVolt
11.
12.     voltInDecimal = int(combinedVolt,16)
13.     #print "The voltInDecimal = %s:  \n" % voltInDecimal
14.
15.     voltMeasured = float(voltInDecimal / 100)
16.     finalVolt = (voltMeasured / Ratio)
17.
18.     #print "The finalVolt is = %s:  \n" % finalVolt
19.     #print "Done temperature calculation!\n"
20.
21.     return finalVolt
```

```
1. #Function record.py
2. def recordV(voltSlave1, voltSlave2, voltSlave3, voltSlave4, vo
   ltSlave5, voltSlave6, voltSlave7, voltSlave8, voltSlave9, volt
   Slave10, voltSlave11, voltSlave12):
3.
4.     import os
5.     import time
6.     import datetime
7.
8.     #print "You are now in recordT"
9.     file = open("/home/pi/Desktop/dataVolt.csv","a")
10.     i = 0
```

```
11.
12.     if os.stat("/home/pi/Desktop/dataVolt.csv").st_size == 0:
13.         file.write("Time,Sensor1,Sensor2,Sensor3,Sensor4,Senso
    r5,Sensor6,Sensor7,Sensor8,Sensor9,Sensor10,Sensor11,Sensor12\
    n")
14.
15.     while(i < 1):
16.         timeRec = datetime.datetime.now().strftime("%Y-%m-%d_%
    H:%M:%S")
17.         file.write(str(timeRec)+","+str(voltSlave1)+","+str(vo
    ltSlave2)+","+str(voltSlave3)+","+str(voltSlave4)+","+str(volt
    Slave5)+","+str(voltSlave6)+","+str(voltSlave7)+","+str(voltSl
    ave8)+","+str(voltSlave9)+","+str(voltSlave10)+","+str(voltSla
    ve11)+","+str(voltSlave12)+"\n")
18.         i = i + 1
19.         file.close
20.     #print "Printing Done.\n"
```

```
1. #GPSread.py
2. #! /usr/bin/python
3. # License: GPL 2.0
4.
5. import os
6. from gps import *
7. from time import *
8. import time
9. import threading
10. import datetime
11. import math
12.
13. gpsd = None #seting the global variable
14. utcOld = 0
15. Result = 0
16. k = 0
17.
18. os.system('clear') #clear the terminal (optional)
19.
20. class GpsPoller(threading.Thread):
21.   def __init__(self):
22.     threading.Thread.__init__(self)
23.     global gpsd #bring gpsd in scope
24.     gpsd = gps(mode=WATCH_ENABLE) #starting the stream of info
25.     self.current_value = None
26.     self.running = True #setting the thread running to true
27.
28.   def run(self):
29.     global gpsd
30.     while gpsp.running:
31.       gpsd.next() #this will continue to loop and grab EACH se
    t of gpsd info to clear the buffer
```

```
32.
33. def isNum(num):
34.     return num != num
35.
36. if __name__ == '__main__':
37.   gpsp = GpsPoller() # create the thread
38.   try:
39.     gpsp.start() # start it up
40.     while True:
41.       #It may take a second or two to get good data
42.       #print gpsd.fix.latitude,', ',gpsd.fix.longitude,'  Time
    : ',gpsd.utc
43.
44.       os.system('clear')
45.
46.       print
47.       print ' GPS reading'
48.       print '----------------------------------------'
49.       print 'latitude    ' , gpsd.fix.latitude
50.       print 'longitude   ' , gpsd.fix.longitude
51.       print 'time utc    ' , gpsd.utc,' + ', gpsd.fix.time
52.       print 'altitude (m)' , gpsd.fix.altitude
53.       print 'eps         ' , gpsd.fix.eps
54.       print 'epx         ' , gpsd.fix.epx
55.       print 'epv         ' , gpsd.fix.epv
56.       print 'ept         ' , gpsd.fix.ept
57.       print 'speed (m/s) ' , gpsd.fix.speed
58.       print 'climb       ' , gpsd.fix.climb
59.       print 'track       ' , gpsd.fix.track
60.       print 'mode        ' , gpsd.fix.mode
61.       print
62.       print 'sats        ' , gpsd.satellites
63.
64.       time.sleep(0.5) #set to delay
65.
66.       file = open("/home/pi/Desktop/dataGPS.csv","a")
67.       i = 0
68.
69.       if os.stat("/home/pi/Desktop/dataGPS.csv").st_size == 0:

70.         file.write("Time,Latitude,Longitude,Altitude(m),Speed(
    kph),Climb(m/min),Track,GPS Mode,Satellites\n")
71.
72.       while(i < 1):
73.         timeRec = datetime.datetime.now().strftime("%Y-%m-%d_%
    H:%M:%S")
74.         file.write(str(timeRec)+","+str(gpsd.fix.latitude)+","
    +str(gpsd.fix.longitude)+","+str(gpsd.fix.altitude)+","+str(gp
    sd.fix.speed)+","+str(gpsd.fix.climb)+","+str(gpsd.fix.track)+
    ","+str(gpsd.fix.mode)+"\n")
75.         i = i + 1
76.         file.close
77.
78.
```

```
  except (KeyboardInterrupt, SystemExit): #when ctrl+c is pres
sed
    print "\nKilling Thread..."
    gpsp.running = False
    gpsp.join() # wait for the thread to finish what it's doin
g
  print "Done.\nExiting."
```

```
#Current Sensor Interfacing and Logging
#LoopingForTesting.py
import ADC
import recordCurrent
import time

cycleValue = 1
while cycleValue < 5800:
    current = ADC.ADC()
    recordCurrent.recordCurrent(current)
    print "Cycle %s is done \n\n\n" %cycleValue
    cycleValue = cycleValue + 1
    #time.sleep(0.5)
```

```
#Function ADC.py
def ADC():

    import spidev
    import time
    import os

    print "You are now in ADC"
    MaxSensorAmp = 600
    LinearlityError = 1.5
    Ipn = 400

    # Initialize SPI bus
    spi = spidev.SpiDev()
    spi.open(0,0)

    # Obtain reference voltage of sensor at 0A
    channel = 1  # Channel 1 is the Reference Voltage
    spidata = spi.xfer2([1, (2+channel) << 6, 0])
    print("Raw ADC:      {}".format(spidata))
    channeldataRef = ((spidata[1] & 31) << 6) + (spidata[2] >>
 2)
    voltageRef = ((channeldataRef * 4.35) / 1024) # 4.35V supp
lied by RaspberryPi
    print("Data ref C1(dec)    {}".format(channeldataRef))
    print("Voltage ref(V): {}".format(voltageRef))

```

```
26.
27.     # Obtain output voltage of sensor (without compensation)
28.     channel = 0  # Channel 0 is the Measured Voltage
29.     spidata = spi.xfer2([1, (2+channel) << 6, 0])
30.     print("Raw ADC:      {}".format(spidata))
31.     channeldata = ((spidata[1] & 31) << 6) + (spidata[2] >> 2)

32.     voltage = ((channeldata * 4.35) / 1024) # 4.35V supplied b
   y RaspberryPi
33.     print("Data C0 (dec)    {}".format(channeldata))
34.     print("Voltage (V): {}".format(voltage))
35.
36.     current = (voltage - voltageRef) / 0.003333
37.     print("Current (A): {}".format(current))
38.
39.     return current
```

```
1. #Function recordCurrent.py
2. def recordCurrent(current):
3.
4.     import os
5.     import time
6.     import datetime
7.     from sympy import*
8.
9.     print "You are now in recordCurrent"
10.     file = open("/home/pi/Desktop/dataCurrent.csv","a")
11.     i = 0
12.
13.     t = Symbol('t')
14.     start_time = time.time()
15.     j = current
16.     end_time = (time.time()- start_time) / 60
17.     totalCharge = integrate(j, (t,0,end_time))
18.
19.     if os.stat("/home/pi/Desktop/dataTemp.csv").st_size == 0:

20.         file.write("Time,Current,Total Charge")
21.     while(i < 1):
22.         timeRec = datetime.datetime.now().strftime("%Y-%m-%d_%
   H:%M:%S")
23.         file.write(str(timeRec)+","+str(current)+str(totalChar
   ge)+"\n")
24.         i = i + 1
25.         file.close
26.     print "Printing Done.\n"
```

```
1. import time
2. from sympy import*
3. t = Symbol('t')
```

```
start_time = time.time()
i = 20   # i = total current
end_time = (time.time()- start_time)/60
totalcharge = integrate(i, (t,0,end_time))
print totalcharge
```

```
#CANBUS interfacing and logging
#Data_CAN.py
#!/usr/bin/env python
#!/bin/bash

import subprocess
import time
import datetime
import os
import RPi.GPIO as GPIO

GPIO.setmode(GPIO.BCM)
GPIO.setwarnings(False)
GPIO.cleanup()
GPIO.setup(17,GPIO.OUT)
GPIO.output(17,GPIO.LOW)
GPIO.setup(27,GPIO.OUT)
GPIO.output(27,GPIO.LOW)
GPIO.setup(22,GPIO.OUT)
GPIO.output(22,GPIO.LOW)
GPIO.setup(24,GPIO.OUT)
GPIO.output(24,GPIO.LOW)

log="/home/pi/Desktop/dataCan.csv"
bashCommand1= "sudo ./slcan_attach -f -s4 -o /dev/ttyACM0"
bashCommand2="sudo ./slcand ttyACM0 slcan0"
bashCommand3="sudo ifconfig slcan0 up"
bashCommand4="sudo ./candump slcan0"
cycle = 0

def runBash(exe, count):
    process = subprocess.Popen(exe.split(), cwd='/can-
utils', stdout=subprocess.PIPE, stderr=subprocess.STDOUT)
    print "executing now"
    while(True):
        retcode = process.poll()  #returns None while subproce
ss is running
        line = process.stdout.readline()
        yield line
        count = count + 1
        print "%d \n" %count
        if(retcode is not None):
            break

for line in runBash(bashCommand1, cycle):
    print line
```

```
45. for line in runBash(bashCommand2, cycle):
46.     print line
47. for line in runBash(bashCommand3, cycle):
48.     print line
49. for line in runBash(bashCommand4, cycle):
50.         print line
51.         timeRec = datetime.datetime.now().strftime("%Y-%m-%d_%
    H:%M:%S")
52.         line = line.strip()
53.         line = line.split()
54.
55.         i = 0
56.         j = 0
57.         m = 0
58.         n = 0
59.         address = int(line[1],10)
60.         if address == 601:
61.             print "Recording dataCAN1./n"
62.             file = open("/home/pi/Desktop/dataCAN1.csv","a")
63.
64.             if os.stat("/home/pi/Desktop/dataCAN1.csv").st_siz
    e == 0:
65.                 file.write("Time, Interface, Address, Bits, Mo
    tor RPM, Motor Temperature, Controller Temperature, RMS Curren
    t, Battery Voltage\n")
66.
67.             while(i < 1):
68.                 rpm = line[3] + line[4]
69.                 rpm = int(rpm,16)
70.                 motorTemp = int(line[5],16)
71.                 conTemp = int(line[6],16)
72.                 current = line[7] + line[8]
73.                 current = (int(current,16))*0.1
74.                 battVolt = line[9] + line[10]
75.                 battVolt = (int(battVolt,16))*0.1
76.                 file.write(str(timeRec)+","+str(line[0])+","+s
    tr(line[1])+","+str(line[2])+","+str(rpm)+","+str(motorTemp)+"
    ,"+str(conTemp)+","+str(current)+","+str(battVolt)+"\n")
77.                 file.close
78.                 break
79.
80.         else:
81.             print "Recording dataCAN2./n"
82.             file = open("/home/pi/Desktop/dataCAN2.csv","a")
83.
84.             if os.stat("/home/pi/Desktop/dataCAN2.csv").st_siz
    e == 0:
85.                 file.write("Time, Interface, Address, Bits, St
    ator Frequency, Primary Controller Fault, Secondary Controller
     Fault, Throttle Input Percentage, Brake Input Percentage, Sys
    tem State\n")
86.
87.             while(i < 1):
88.                 statorF = line[3] + line[4]
```

```
89.                 statorF = int(statorF,16)
90.                 if statorF > 1:
91.                     faultP = int(line [5],16)
92.                     print faultP
93.                     faultS = int((line [6]),16)
94.                     print faultS
95.                     throttle = int(line[7],16)
96.                     brake = int(line[8],16)
97.                     system = int(line[9],16)
98.                     print system
99.                     file.write(str(timeRec)+","+str(line[0])+"
   ,"+str(line[1])+","+str(line[2])+","+str(statorF)+","+str(faul
   tP)+","+str(faultS)+","+str(throttle)+","+str(brake)+","+str(s
   ystem)+"\n")
100.                            file.close
101.                            if faultP > 0:
102.                                print "WARNING: CONTROLLER FAULT
    DETECTED!\n Initiating EMERGENCY STOP!"
103.                                GPIO.output(17,GPIO.HIGH)
104.                                time.sleep(0.1)
105.                                GPIO.output(17,GPIO.LOW)
106.                                if faultP < 10:
107.                                    while (j != faultP):
108.                                        GPIO.output(24,GPIO.HIGH
   )
109.                                        time.sleep(0.5)
110.                                        GPIO.output(24,GPIO.LOW)

111.                                        time.sleep(0.5)
112.                                        j = j + 1
113.                                    if (j == faultP):
114.                                        n = 0
115.                                        time.sleep(3)
116.                                        break
117.                                else:
118.                                    faultP = str(faultP)
119.                                    faultS = str(faultS)
120.                                    faultP1 = faultP[0]
121.                                    faultP2 = faultP[1]
122.                                    while (m != faultP1):
123.                                        GPIO.output(24,GPIO.HIGH
   )
124.                                        time.sleep(0.5)
125.                                        GPIO.output(24,GPIO.LOW)

126.                                        m = m + 1
127.                                    if (m == faultP1):
128.                                        m = 0
129.                                        GPIO.output(24,GPIO.HIGH
   )
130.                                        time.sleep(1.5)
131.                                        GPIO.output(24,GPIO.LOW)

132.                                        break
```

```
                                    while (n != faultP2):
                                        GPIO.output(24,GPIO.HIGH
    )
                                        time.sleep(0.5)
                                        GPIO.output(24,GPIO.LOW)

                                        m = n + 1
                                    if (n == faultP2):
                                        n = 0
                                        time.sleep(3)
                                        break

                                if system == 10 :
                                    print "ECO MODE"
                                    GPIO.output(27,GPIO.HIGH)
                                    time.sleep(0.5)
                                    GPIO.output(27,GPIO.LOW)

                                if system == 42 :
                                    print "REGENERATIVE MODE"
                                    GPIO.output(22,GPIO.HIGH)
                                    time.sleep(0.5)
                                    GPIO.output(22,GPIO.LOW)
                                break
```

```
#Auto Run Script on Boot
import time
import threading
import os

def startprgm(i):
    print "Running thread %d" % i
    if (i == 0):
    print('Running: Data_CANp.py')
    os.system("sudo python /home/pi/Desktop/FINAL_CODES/CANBUS
/Data_CANp.py")
    elif (i == 1):
    print('Running: GPSread.py')
    os.system("sudo python /home/pi/Desktop/FINAL_CODES/GPS/GP
Sread.py")
    elif (i == 2):
    print('Running: Current-LoopingForTesting.py')
    os.system("sudo python /home/pi/Desktop/FINAL_CODES/Curren
t/LoopingForTesting.py")
    elif (i == 3):
    print('Running: TempVolt-LoopingForTesting.py')
    os.system("sudo python /home/pi/Desktop/FINAL_CODES/Loopin
gForTesting.py")
    elif (i == 4):
        print('Running: shutdown.py')
        os.system("sudo python /home/pi/Desktop/Scripts/shutdo
wn_pi.py")

```

```
      else:
        pass

for i in range(5):
    t = threading.Thread(target=startprgm, args=(i,))
    t.start()
```

```
#Shut Down Script
#shutdown_pi.py
#!/bin/python
# Simple script for shutting down the Raspberry Pi at the pres
s of a button.

import RPi.GPIO as GPIO
import time
import os

# Use the Broadcom SOC Pin numbers
# Setup the Pin with Internal pullups enabled and PIN in readi
ng mode.
GPIO.setmode(GPIO.BCM)
GPIO.setwarnings(False)
GPIO.setup(24,GPIO.OUT)
GPIO.setup(23, GPIO.IN, pull_up_down = GPIO.PUD_UP)

# Add function on what to do when the button is pressed
def Shutdown(channel):
    os.system("sudo shutdown -h now")
    GPIO.output(24,GPIO.HIGH)
    time.sleep(2)
    GPIO.output(24,GPIO.LOW)
    time.sleep(2)

# Add function to execute when the button pressed event happen
s
GPIO.output(24,GPIO.HIGH)
time.sleep(0.1)
GPIO.output(24,GPIO.LOW)
time.sleep(0.1)
GPIO.output(24,GPIO.HIGH)
time.sleep(0.1)
GPIO.output(24,GPIO.LOW)
time.sleep(0.1)
GPIO.output(24,GPIO.HIGH)
time.sleep(0.1)
GPIO.output(24,GPIO.LOW)
GPIO.add_event_detect(23, GPIO.FALLING, callback = Shutdown, b
ouncetime = 2000)

# Now wait!
while 1:
    time.sleep(1)
```