A REMOTE VEHICLE HEALTH MONITORING SYSTEM: A CASE STUDY OF THE KAYOOLA ELECTRIC VEHICLES

Immaculate Kamusiime

A Project Report Submitted in Partial Fulfillment of the Requirements of the Award the Degree of Master of Science in Embedded and Mobile Systems of the Nelson Mandela African Institution of Science and Technology

Arusha, Tanzania

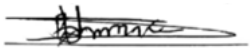
July, 2023

ABSTRACT

Due to their complex hardware and software, vehicle systems are challenging to monitor and maintain. As a result, the absence of proper vehicle health monitoring systems leads to several issues, including resource waste, costly maintenance, and frequent breakdowns. Therefore, vehicle manufacturers and owners must prioritize implementing such systems to mitigate these negative outcomes. This project report focuses on a remote vehicle health monitoring system explicitly developed for Kiira Motors Corporation's (KMC's) electric vehicles made in Uganda. This research project involved interpreting real-time data transmitted through the vehicle's Controller Area Network (CAN) into a human-readable format using the developed embedded device and a web application to notify technical operators of the vehicle's health status remotely. The system consists of a wireless link between the embedded device and the web application and has been integrated into Kiira EV, one of KMC's concept vehicles. This embedded device is attached to the vehicles' CAN data hub through CAN-high and CAN-low communication lines to receive live data through electric signals generated by various sensors located throughout the vehicle. The developed system was integrated into the Kiira EV, and five critical vehicle parameters, including pack voltage, motor temperature, motor speed, pack current, and vehicle location, were remotely monitored using the developed web application. The developed system enables the technical vehicle operator to remotely track, monitor, and receive notifications on the general health status of the Kiira EV based on the streamed live CAN data.

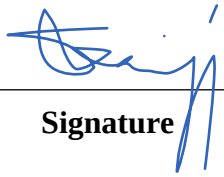
DECLARATION

I, **Immaculate Kamusiime**, declare to the Senate of the Nelson Mandela African Institution of Science and Technology that this project report is my original work and that it has neither been submitted nor concurrently submitted for a degree award in any other institution.

Immaculate Kamusiime		17th July 2023
Name of Candidate	**Signature**	**Date**

The above declaration is confirmed by:

Dr. Emmanuel Masabo		18th July 2023
Name of Supervisor 1	**Signature**	**Date**

Prof. Shubi Kaijage		18-07-2023
Name of Supervisor 2	**Signature**	**Date**

COPYRIGHT

CERTIFICATION

The undersigned certify that they have read and, with this recommendation for acceptance by the Nelson Mandela African Institution of Science and Technology, a project report titled ***"A Remote Vehicle Health Monitoring System with a Case Study of Kayoola Electric Vehicles"*** in partial fulfilment of the requirements for the degree of Master of Science in Embedded and Mobile Systems of the Nelson Mandela African Institution of Science and Technology.

Dr. Emmanuel Masabo		18th July 2023
Name of Supervisor 1	**Signature**	**Date**

Prof. Shubi Kaijage		18-07-2023
Name of Supervisor 2	**Signature**	**Date**

ACKNOWLEDGMENTS

First and foremost, I thank Almighty God for blessing me and allowing me to pursue a master's degree program at the Nelson Mandela African Institution of Science and Technology (NM-AIST). Furthermore, I am forever grateful for the courage, strength, and knowledge I gained throughout my studies.

I would like to express my heartfelt gratitude to the Center of Excellence for ICT in East Africa (CENIT@EA) for providing financial assistance during my two-year studies at the Nelson Mandela African Institution of Science and Technology (NM-AIST).

I would like to thank my academic supervisors, Dr. Emmanuel Masabo and Prof. Shubi Kaijage, for their unending support and guidance during my internship for the research project.

I want to thank my industrial supervisor Mr. Fred Matovu for his continuous guidance and support while at Kiira Motors Corporation (KMC) in Uganda as I pursued my graduate research project. I am very grateful.

I am also thankful to the management of Kiira Motors Corporation for opening their doors for me to do my internship and project at the organization. It would be robbery if I did not appreciate the staff at Kiira Motors Corporation for providing a conducive environment to learn about the automotive industry and complete my research project in time.

DEDICATION

The Almighty God, who began this noble endeavour and saw it through to completion, is honoured with this project report.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

LIST OF APPENDICES

LIST OF ABBREVIATIONS

AC	Alternating Current
API	Application Programming Interface
BMS	Battery Management System
CAN	Controller Area Network
Ctrl + N	Control Plus N
DBC	Database
DC	Direct Current
DP	Data Page
DRF	Django Rest Framework
DTC	Diagnostic Trouble Codes
EDA	Exploratory Data Analysis
EPCU	Electric Power Control Unit
EVs	Electric Vehicles
GHz	Giga Hertz
GPS	Global Positioning System
HLP	Higher Layer Protocol
ICE	Internal Combustion Engines
IDE	Integrated Development Environment
KMC	Kiira Motors Corporation
LE	Little Edian
LED	Light-Emitting Diode
NM-AIST	Nelson Mandela African Institution of Science and Technology
OBD	On-Board Diagnostic
OCV	Open Circuit Voltage

OEMs	Original Equipment Manufacturer
P	Priority
PC	Personal Computer
PCA	Principal Component Analysis
PCB	Printed Circuit Board
PDU	Protocol Data Unit
PGN	Parameter Group Number
R	Reserved
SA	Source Address
SOC	State of Charge
SOE	Society of Automotive Engineers
SOH	State of Health
SPN	Suspect Parameter number
TA	Technical Administrator
TP	Transport Protocol
UP	Uganda Police
V	Volt
VCU	Vehicle Control Unit
XML	Extensive Markup Language
XP	Extreme Programming
YAML	Yet Another Markup Language

CHAPTER ONE

INTRODUCTION

1.1 Background of the Problem

Kiira Motors Corporation (KMC) is a State Enterprise created to promote value addition in Uganda's emerging motor vehicle industry through technology transfer, contract manufacturing, and supply chain localization. KMCis located at Plot 13 Kimera Rd, Kampala, Uganda. Its mission is to improve Uganda through automotive technology and create long-term technical partnerships to strengthen its fundamental competencies for developing, manufacturing, and selling automobiles and their components in Africa. This tactical move supports the industrialization program to help Uganda become an upper-middle-income country by 2040. The 4+1 pillars of People, Plant, Products, Mobility Infrastructure, and Policy are the foundation of the Strategic Technology Partnership(s), designed and influenced by the facts and features of the target market in Uganda and Africa. The foundation for the sustainability of KMC operations is the efficient use of facilities, infrastructure, and resources, along with the appropriate use of Industry 4.0+ tools and technology in a circular economy framework. KMC is owned and managed by the Government of the Republic of Uganda, represented by the Ministry for Science, Technology, and Innovation, where the Office of the President (96%) and Makerere University (4%) are the company's shareholders (kiiramotors, 2022).

Managing and monitoring vehicle health is another area that companies and organizations must carefully look into because failure to implement successful vehicle management and monitoring systems results in wasted resources, expensive and wasteful maintenance, and several breakdowns nearly every day.

Due to the absence of proper vehicle health monitoring systems, Uganda Police (UP) reports that driving vehicles in dangerous mechanical conditions is one of the major contributors to road vehicle failures and accidents, resulting in numerous road fatalities (The Uganda Radio Network, 2022). UP also reports an accident along the express highway; its causes were unclear, which calls for awareness about the vehicle’s health status to keep them (Force, 2022). As shown in Fig. 1, Uganda police continued registering more vehicle failures on the road, encouraging everyone to ensure the suitability and worthiness of vehicles. Most of these vehicles are serviced and maintained on a calendar-based schedule or as needed when the

problem occurs, thus not solving the problem. This typical maintenance technique utilized in the automotive industry to resolve the issue and serve as a cost-effective approach is reactive, which reduces the vehicle's lifetime and costs money (Force, 2022b).

Figure 1: Vehicle failures on the road (Africa-press, 2022)

To remedy the situation and address the problem at hand, implementing a remote vehicle health monitoring system to alert and provide the owners with the status of the health and performance of the vehicles is crucial at this point. The remote vehicle health monitoring system elaborated in this project report assists vehicle owners in keeping track and quickly identifying any technical faults with the vehicles before they fail using real-time data, thus allowing the user to work more proactively and save time. The pack voltage, pack current, motor speed, motor temperature, and vehicle location are the five significant parameters of Kiira EV that were the point focus for this research project. Furthermore, the method is being developed to increase vehicle uptime and ensure that vehicles are always roadworthy.

1.2 Statement of the Problem

The Kiira Motors Corporation does not have a remote centralized vehicle health monitoring system to inform its technical operators of the vehicle's health status and available faults. As a result, random sampling is the only way to troubleshoot vehicles and perform maintenance. However, this method has resulted in several other issues, including resource wastage, an endless loop of expensive maintenance, and frequent breakdowns. To address these concerns, the research project aimed to develop a remote vehicle health monitoring system designed

explicitly for KMC's electric vehicles manufactured in Uganda, which updates the technical vehicle operators on the vehicle's health statuses.

1.3 Rationale of the Study

Kiira Motors Corporation (automotive mobility enterprise) was founded to manufacture and sell automobiles in Africa; therefore, the study project justifies prioritizing adopting remote vehicle health monitoring systems to reduce resource waste, costly maintenance, and frequent breakdowns. However, the problem statement underlined Kiira Motors Corporation's (KMC) lack of a centralized remote vehicle health monitoring system, which resulted in random sampling for troubleshooting and maintenance, with detrimental consequences. As a result, the research project aimed to increase vehicle reliability by better understanding their current health status by developing a remote vehicle health monitoring system specifically developed for KMC's Uganda-made electric vehicles.

1.4 Objectives of the Study

1.4.1 Main Objective

To develop the remote vehicle health monitoring system for Kayoola electric vehicles.

1.4.2 Specific Objectives

(i) To identify requirements specifications of the remote vehicle health monitoring system for Kayoola electric vehicles.

(ii) To develop the prototype and web application of the remote vehicle health monitoring system for Kayoola electric vehicles.

(iii) To validate the remote vehicle health monitoring system for Kayoola electric vehicles.

1.5 Research Questions

(i) What are the requirements specifications for developing the remote vehicle health monitoring system for Kayoola electric vehicles?

(ii) What methods and materials can be used to develop the remote vehicle health monitoring system for Kayoola electric vehicles?

(iii) Did the developed prototype give the expected outcome?

1.6 Significance of the Study

The study aims to develop a remote vehicle health monitoring system that enables real-time data collection to improve vehicle reliability and availability. The study centres on the Kayoola Electric Buses manufactured by Kiira Motors Corporation for public transport. The system's unique contribution is that it interprets data into a human-readable format and provides notifications about the vehicle's health status in advance. The system is cost-efficient and reduces downtime, essential for effective fleet management. The system's significance lies in acquiring raw data from the Kayoola electric vehicle and transmitting it wirelessly to the web application for remote monitoring, saving time and money for fleet managers, improving vehicle safety and dependability, and reducing vehicle breakdowns and maintenance costs.

1.7 Delineation of the Study

This project involves the development of an embedded system prototype integrated into a web application for remote vehicle health monitoring. The project aims to increase safety, reduce maintenance costs, and improve performance by collecting diagnostic data from the vehicle's CAN communication network and providing real-time health status updates. Thorough testing, validation, and implementation of advanced cybersecurity measures are crucial considerations. The project's contribution lies in offering a practical solution for remote monitoring of electric vehicles' health status, advancing the field of IoT and smart transportation systems while addressing the specific needs of vehicle owners and automotive companies.

CHAPTER TWO

LITERATURE REVIEW

2.1 Related Works

The paper discusses the emergence of the Internet of Things (IoT) technology in the automotive industry, transforming vehicles into an entire ecosystem of connected IoT services. It shows that IoT has enabled the development of intelligent vehicle diagnostic systems, allowing for vehicle and fleet tracking, fault detection, and analysis of driver behaviour, among other operational needs. It also discusses that IoT protocols have facilitated remote monitoring and control over an existing network, improving system accuracy, efficiency, and security. Overall, this paper highlights the potential benefits of IoT-based systems in enhancing safety, security, and product offerings in the vehicular sector (Singh *et al.*, 2021). However, this research paper lacks specific details on how the IoT-based systems improved the current structure of the automotive industry, which could limit its usefulness for electric vehicle manufacturers seeking more in-depth information for adopting remote monitoring.

Momin *et al.* (2020) wrote a review paper on using a vehicle health monitoring system to help notify the owner about issues such as tire pressure, exhaust gas quality, brake wear, oil leaks, engine overheating, and fuel flow blockages. Various sensors, such as tire pressure monitoring, gas, flow, and temperature sensors, can provide a real-time display on an LCD unit connected to an Arduino microcontroller (Momin *et al.*, 2020). The review paper was exclusively designed for engine cars, employs LCD to display results, and does not provide a way to monitor the vehicle's health remotely.

Modern vehicles typically have an OBDII port to provide diagnostic data using an ELM327. This Microcontroller can give information on various vehicle parameters, including speed, temperature, battery level, and error codes for fault detection. However, it is impractical to store and analyze all of the collected data on a mobile phone. This research aimed to develop an algorithm to analyze the data and provide statistics on whether the currently measured values are within suitable ranges. Furthermore, monitoring the data and examining any alarming readings would reduce the probability of major faults occurring and provide information about the occurrence of faults (Bánhelyi & Szabó, 2020). However, this research lacks specific details and context on developing the application and algorithm for limited hardware resources. It also

does not mention the scope or focus of the research, which can make it difficult to assess its relevance and contribution to the field.

This paper reviews the importance and necessity of reasoning applications in the Aerospace Integrated Vehicle Health Management (IVHM) field. Integrated Vehicle Health Management requires a fully functional system to optimize Condition Based Maintenance (CBM) and reduce unplanned maintenance costs. To achieve this, supporting technologies like AI and sensor technology are needed. The paper focuses on reasoning technology and explores how it has helped IVHM in the past, reviewing various reasoning strategies, systems, and architectures. Shortcomings in the IVHM field, particularly in vehicle-level health monitoring, are discussed, along with how reasoning can address some of them. Finally, the paper highlights the challenges in developing reasoning systems for vehicle-level health monitoring and proposes ways to mitigate them (Ezhilarasu *et al.,* 2019). The limitation of this research paper is that it only discusses the importance and necessity of reasoning applications in the field of Aerospace Integrated Vehicle Health Management (IVHM) without providing specific examples or results of practical applications of reasoning technology.

The automotive industry is increasingly focusing on predictive maintenance to prevent vehicle failures. Despite limited sensor availability and design constraints, machine learning techniques can be used to analyze sensor data for failure prediction. This study presents an approach for fault prediction of four main vehicle subsystems: fuel, ignition, exhaust, and cooling. Sensor data is collected from faulty and normal conditions and analyzed using classifiers such as Decision Tree, Support Vector Machine, K Nearest Neighbor, and Random Forest. The approach aims to increase vehicle up-time and was tested on 70 Toyota Corolla vehicles. Classifier accuracy is compared using Receiver Operating Characteristics (ROC) curves (Shafi *et al.,* 2018). However, this study focuses only on a specific type of diesel vehicle (Toyota Corolla), so the generalizability of the findings to other vehicle models is unclear. Also, it doesn't show how the results are monitored and relayed to the end user of the vehicle.

Perisoara *et al.* (2018) designed and deployed a pilot platform for electric vehicle monitoring via the Internet. The platform contains a vehicle-based embedded device (EVCANMon) and a web-server software application (EVWebMon). This collects data such as speed, location, motor temperature and speed, voltage and battery current, controller temperature, and fault codes SoC from various ECUs via the vehicle's Controller Area Network (CAN) and sends it using GPRS from a GSM local network. In addition, the software application allows the saving

of the parameters being monitored in a MySQL database and the visualization and analysis of data using interactive maps and graphs (Perisoara *et al.,* 2018).

Ganesa and Mydhile (2013) wrote this research paper which proposes a remote vehicle monitoring system that addresses the challenges of manual monitoring and user behaviour changes. The proposed method provides real-time monitoring via a web-based application and a novel approach to modelling dynamic behaviours of interacting context-aware web services. To detect changes in the system execution context and determine how the system should behave an advanced self-adapting algorithm is developed. The goal is to provide an adaptive environment that allows the system to change its behaviour depending on the contexts that ensure context-aware web services challenges. However, it only focuses on remote monitoring of vehicle location and user behaviour and does not address how a vehicle's health can be monitored. It also lacks specific research methodology, scope, and results details. Additionally, it is unclear about the specific context and application of the proposed remote vehicle monitoring system.

The paper presents a new vehicle online diagnosis and real-time early warning system based on the On-Board Diagnostics (OBD) system to acquire signals and transmit them to a server via mobile communication. The system aims to provide immediate actions for maintenance engineers once a warning signal is generated, which most operators are not known to take action on. The proposed system includes hardware and software design and implementation, and the test functions fulfil the modern requirements for the vehicle system. As a result, the system improves the maintenance efficiency of vehicles and ensures their health and safety (Lin *et al.,* 2005). However, this research does not provide any information on the effectiveness or accuracy of the system in detecting vehicle system errors or malfunctions. Therefore, whether the proposed system can provide reliable and accurate diagnosis and early warning for modern vehicle systems is unclear.

2.2 Research Gap

The methods, systems, and solutions proposed in the related work can only be used in diesel vehicles but not heavy vehicles (electric vehicles). They use mobile phones, which can only be used by drivers, which take their attention while on the road and end up causing more accidents on the road. The proposed remote vehicle health monitoring system is customized to only Kayoola electric vehicles and can remotely notify the technical operator of critical vehicle

faults. With the same system, the technical operator can add new vehicles not initially in the system and schedule their maintenance using the information fed into it. With the Centralized Vehicle Health Monitoring System, any vehicle fault can be learned earlier, and the technical operator gets notified before the vehicle breakdowns. It is remarkable how this research project routes data to the web application using the Internet of Things (IoT).

2.3 Conceptual Framework

The conceptual framework for this research project focuses on developing and implementing a remote vehicle health monitoring system. It involves an embedded system prototype, web application, data collection from the vehicle's CAN communication network, and desired outcomes of increased safety, reduced maintenance costs, and improved vehicle performance. The embedded system prototype, consisting of Arduino Nano IoT 33 and MCP2515 CAN BUS module, collects diagnostic data from the vehicle's sensors and wirelessly transmits it to the web application for storage and visualization. The objectives include real-time health status updates, predictive maintenance, and optimizing vehicle performance. Testing, validation, and cybersecurity measures are essential considerations while the project contributes to the field of IoT and smart transportation systems.

CHAPTER THREE

MATERIALS AND METHODS

3.1 Study Area

The study and scope are centred around Kayoola electric vehicles manufactured by Kiira Motors Corporation (KMC) for public transport. The concentration is put on monitoring of health statuses of Kayoola electric vehicles remotely, and only five critical vehicle parameters of pack voltage, motor temperature, motor speed, pack current, and vehicle location were monitored.

3.2 System Development Stages

The system development was done through processes in Fig. 2 till its completion. All these processes are dependent on each other for a fully functioning system.

The first involves data acquisition using the developed embedded device integrated into the Kiira EV's CAN network to gather data across the CAN bus's CAN-H and CAN-L communication lines. The embedded device combines hardware and software, has fixed capabilities, and is intended to carry out specific tasks within a broader mechanical or electrical system. It is a standalone unit or a component of a larger system. It is primarily intended for a particular purpose or function within a bigger system (Guru99, 2022). have its own embedded CAN data logger system to allow continuous real-time data collection.

Second, this process is the data analytics stage which involves data processing using different techniques. Finally, the raw data stream is continuously transmitted to the web database for analysis and processing.

Third, it is a web visualisation process demonstrated later in the web application system.

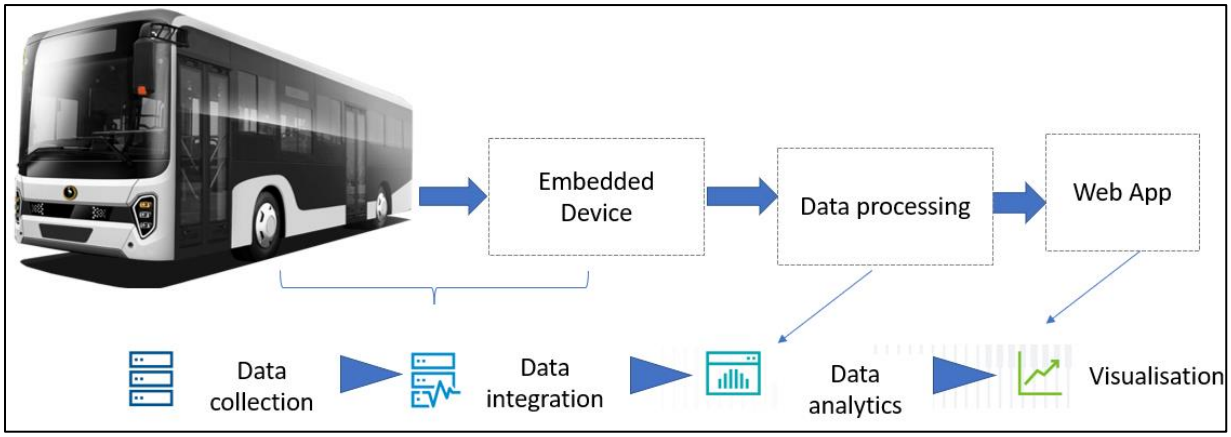


Figure 2: System development processes

3.3 System Context Diagram

Figure 3 is a system context diagram illustrating the operation of the functional blocks of the entire remote vehicle health monitoring system and how they relate. It consists of the embedded system, the CAN bus of the vehicle, the administrator, the system database, and the web application. The embedded system comprises a microcontroller (Arduino Nano 33 IoT) and MCP2514 Controller Area Network (CAN) module. It acquires diagnostic data from the vehicle's OBD II port originating from different sensors and routes it to the web application through a wireless connection for visualization, management, and notification in case of any critical issue to avoid an unexpected vehicle stoppage on the road, thus boosting employee morale and helping the organization fulfil its core goals. In addition, it provides diagnostic information on the VCU, motor controller, DC-DC converter, battery management system, vehicle performance information, maintenance schedules, and energy consumption, which aids in cost management and allows the vehicles to operate for extended periods. The data is retrieved from the CAN port with a microcontroller (Arduino Nano 33IoT) interfaced with the MCP2515 CAN module, which is transmitted to the server for analysis.

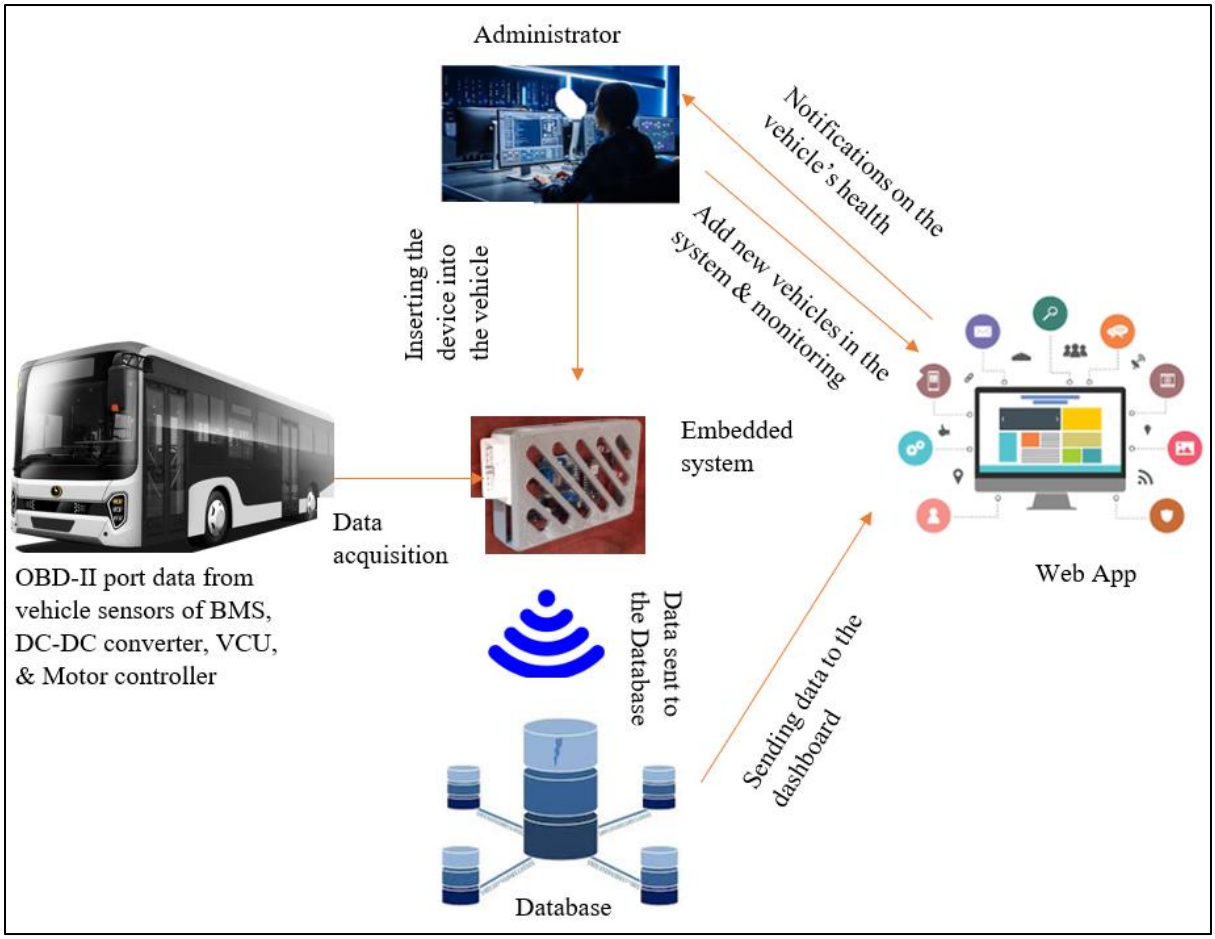


Figure 3: System context diagram

3.4 System Requirements Specifications

The system requirements specifications section addresses the results of research question 1 as stated in specific objective number one. These include both functional requirements and non-functional requirements.

3.4.1 Functional Requirements

(i) The system should be able to monitor the pack voltage

(ii) The system should be able to monitor the pack's current

(iii) The system should be able to monitor the motor temperature

(iv) The system should be able to monitor the motor speed.

(v) The system should be able to track the location of the vehicle.

(vi) The system should be able to collect data on the vehicle of the five parameters monitored.

3.4.2 Non-functional Requirements

(i) The system should be secure and protect sensitive data from unauthorized access.

(ii) The system should be reliable and able to operate continuously without failure.

(iii) The system should be scalable and able to handle many Kayoola electric vehicles.

(iv) The system should be user-friendly and provide a simple and intuitive interface for users to access the data and alerts.

(v) The system should be accessible from any device with an internet connection.

3.5 Materials

This section describes the materials used in project implementation, comprising the software and hardware described in the preceding sections. The first section describes the hardware components and their specifications for developing the remote vehicle health monitoring system. Finally, it describes the software used to program the prototype (embedded device) and the development web application.

3.5.1 Hardware Development Requirements

The key hardware materials used in developing the prototype are described in this section: Arduino Nano IoT 33, MCP2515, and DC to DC step-down converter. It also describes how the hardware components interface with one another and prototype development, simulation, and programming.

(i) Arduino Nano IoT 33

The Arduino Nano IoT 33 is an Internet of Things (IoT) microcontroller board. It uses the SAMD21G18A microprocessor and includes a low-power WiFi module (ESP32). The Arduino IDE can program the board's digital and analog inputs/outputs. The Arduino Nano IoT 33 has a significant advantage over other microcontrollers due to its WiFi functionality, which allows the board to connect to the Internet and communicate with other devices over a wireless network. Because of this functionality, it was an excellent solution for developing a remote vehicle health monitoring system. Furthermore, its tiny form size and low power consumption

were ideal for creating portable and battery-powered embedded devices. Figure 4 illustrates the Arduino Nano IoT 33 used for this project with its pinouts in Fig. 5; its specifications are detailed in Table 1.

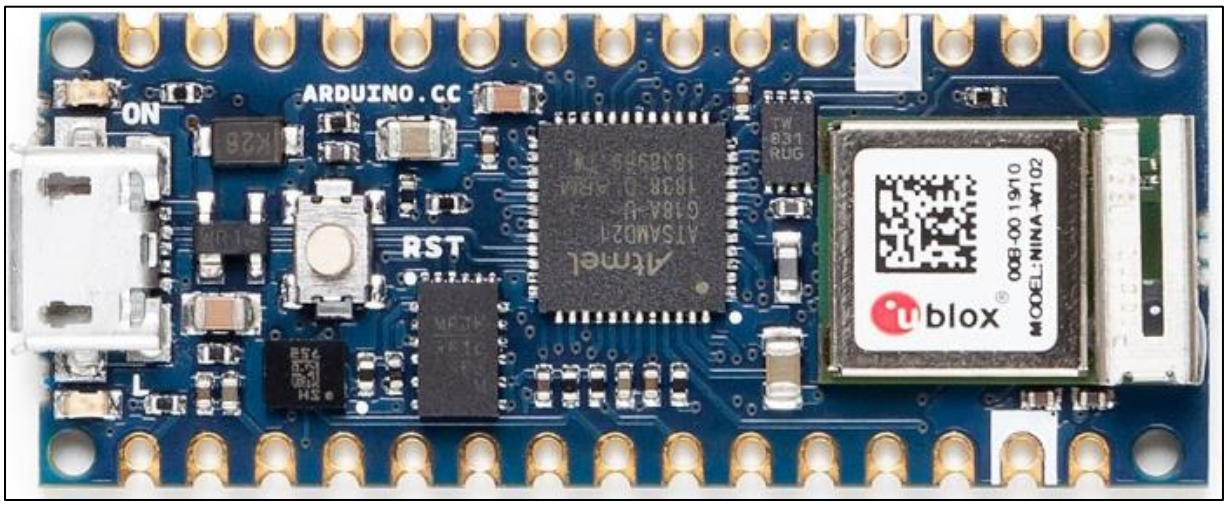


Figure 4: Arduino Nano 33 IoT diagram

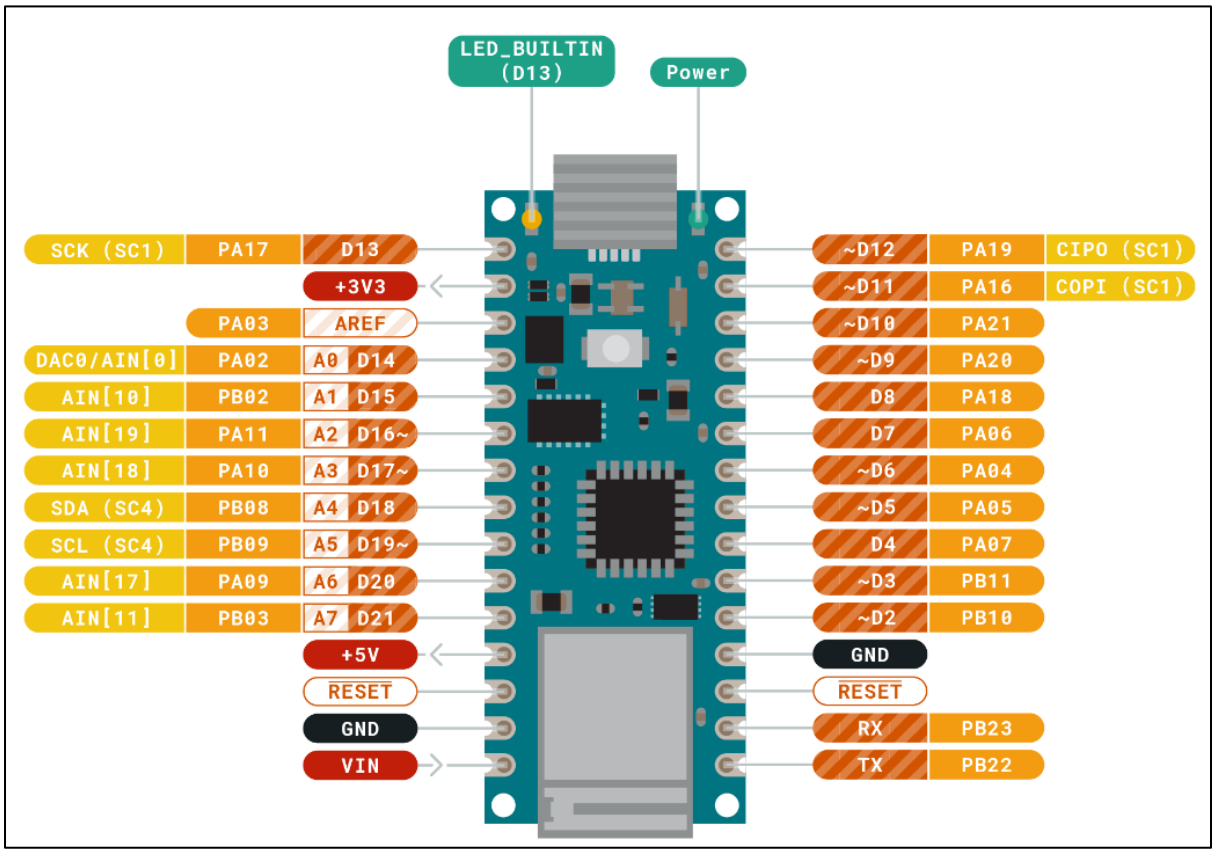


Figure 5: Arduino Nano 33 IoT pinouts (IoT, 2022)

Table 1 depicts the major specifications of the Arduino Nano 33 IoT, which serves as the primary Microcontroller in the created embedded system. This Microcontroller is programmed. It is set up to control the other hardware components of the developed embedded system.

Table 1: Arduino Nano 33 IoT Specifications

S/N	Component	Support	Rating
1	Microcontroller	SAMD21 Cortex-M0+ 32bit low power ARM MCU	N/A
2	Pins	LED pin built-in	13
		I/O digital pin	14
		Input analogue pin	8
3	Connection types	WiFi	Nina W102 uBlox module
		Bluetooth	
4	Sensors	IMU	LSM6DS3
5	Protocols for Communication	UART	Both RX and TX
		I2C	A4 for SDA, A5 for SCL
		SPI	COPI (D11 and D12) and D13 for (SCK). Any GPIO can be used for Chip Select (CS).
6	Power	Voltage (I/O)	It is 3.3V
		Input nominal voltage	It is 5-18V
		DC Current per I/O Pin	It is 7 mA
7	Clock speed	Processor used	SAMD21G18A, 48MHz

(ii) MCP2515 CAN Module

The MCP2515 CAN module is a controller area network (CAN) bus controller module that allows connection between the Microcontroller and the vehicle's CAN communication network. The module serves as an interface between the CAN bus and a microcontroller, allowing the Microcontroller to read data to and from the CAN network. The MCP2515 module shown in Fig. 6 was chosen over other CAN modules because of its simple interface and compatibility with the Arduino Nano IoT 33 microcontroller used in this research project. Furthermore, the module includes an SPI interface, making it simple to connect to

the Microcontroller. Table 2 has the major specifications of the MCP2515 CAN module in Fig. 6.

Figure 6: MCP2515 CAN Module (Microchip, 2019)

Table 2: MCP2515 Specifications

S/N	Characteristic	Rating
1	Communication speed	1Mb/s
2	Bus length	1000 meter
3	Supported frames	Standard, extended and remote
4	Communication protocol	SPI interface
5	Power supply	5V
6	Working current (1 microamp standby current)	5mA
7	Operating temperature	-40 C - +85 C
8	Onboard termination resistance	120 ohms

The MCP2515 CAN module is an SPI-based controller device communicating with various CAN nodes through a CAN bus. The MCP2515 module's block diagram in Fig. 7 shows an SPI interface for communicating with an external microcontroller unit for encoding and decoding CAN messages, a message buffering system for storing incoming and outgoing messages, a clock generator for generating necessary timing signals, and an interrupt control unit for handling various interrupt sources. In addition, the module supports standard and extended data frames and a variety of bit rates, making it an excellent choice for data acquisition from a CAN communication network (CopperHillTechnologies, 2022).

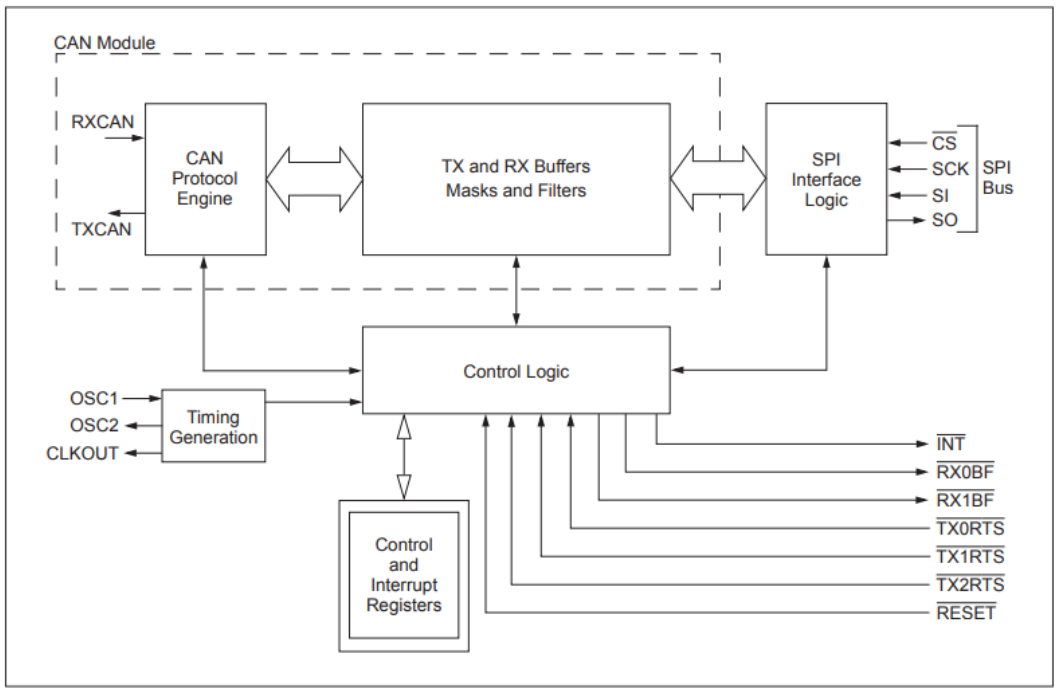


Figure 7: MCP2515 Block Diagram (Microchip, 2019)

(iii) LM2956 DC-DC Step-Down Converter

The Kayoola electric vehicles' electronic components are supplied with 24 volts, and each component down converts it internally to the required power. This LM2956 DC-DC step-down converter in Fig. 8 with specifications in Table 3 was used to convert the 24 volts to 5 volts that supply the developed circuit of the remote vehicle health monitoring system.

Table 3: LM2596 Specifications

S/N	Characteristic	Rating
1	Input-voltage	Ranges from 3V to 40V
2	Output-voltage	Ranges from 1.5V to 35V, but it is (Adjustable)
3	Output-current	Ranges from 2A (min) to 3A (max).
4	Switching-Frequency	150KHz.
5	Operating-temperature	Industrial grade is from -40 to +85
6	Conversion-efficiency	The maximum 92%

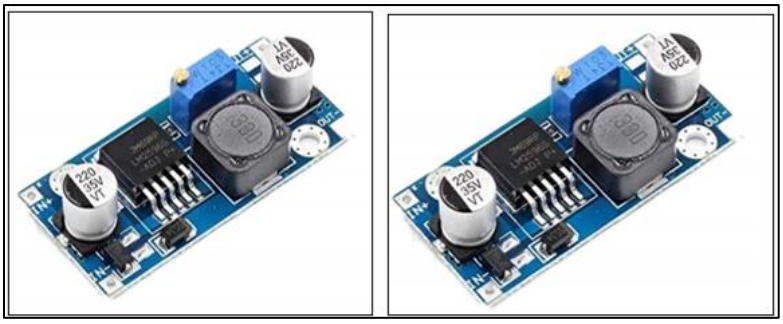


Figure 8: Step-down converter

(iv) Hardware Components Integration

The integration of the hardware components used in the development of the embedded system used in data acquisition from the CAN communication system is depicted in Fig. 9. The Microcontroller (Arduino Nano 33 IoT) is at the heart of the system, connecting to WiFi and the MCP2515 CAN module. To acquire diagnostic data, the MCP2515 features CAN-H and CAN-L lines that link with the CAN-H and CAN-L communication lines of the vehicle's CAN network. In addition, the Microcontroller used provides WiFi connectivity, making it simple to wirelessly transmit data from the CAN communication network to the remote server.

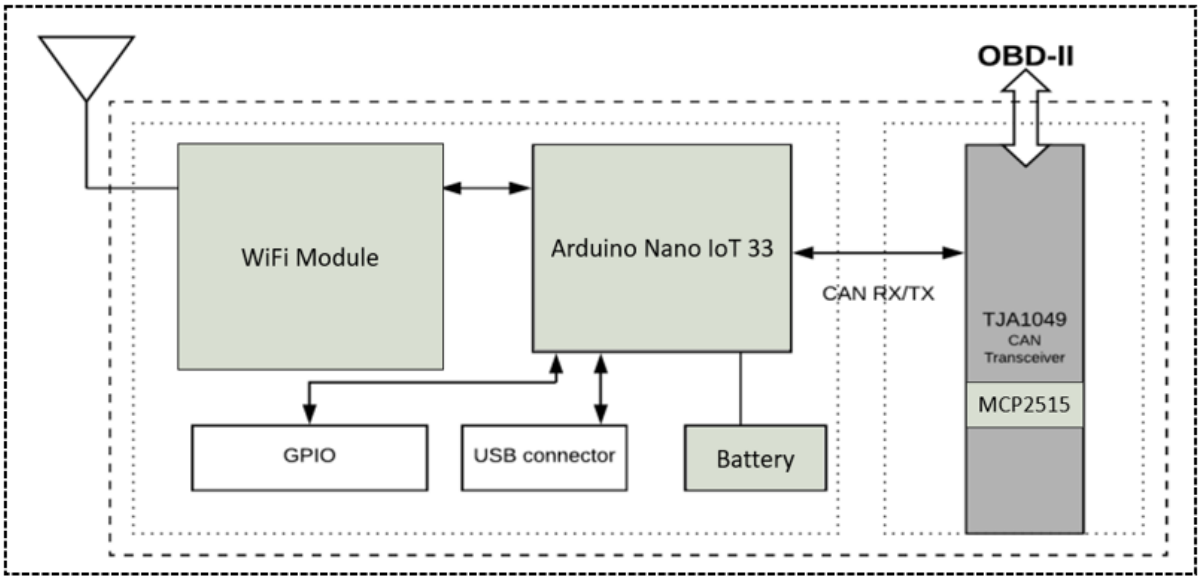


Figure 9: Hardware integration block diagram

(v) Prototype Development Stages

The prototype (embedded device) for data acquisition from the CAN communication network of the Kayoola electric vehicles was designed and developed throughout the four stages detailed in Fig. 10. The four stages of circuit design and simulation, system circuit

demonstration, implementation, and packaging are described in the succeeding sections under the prototype development stages.

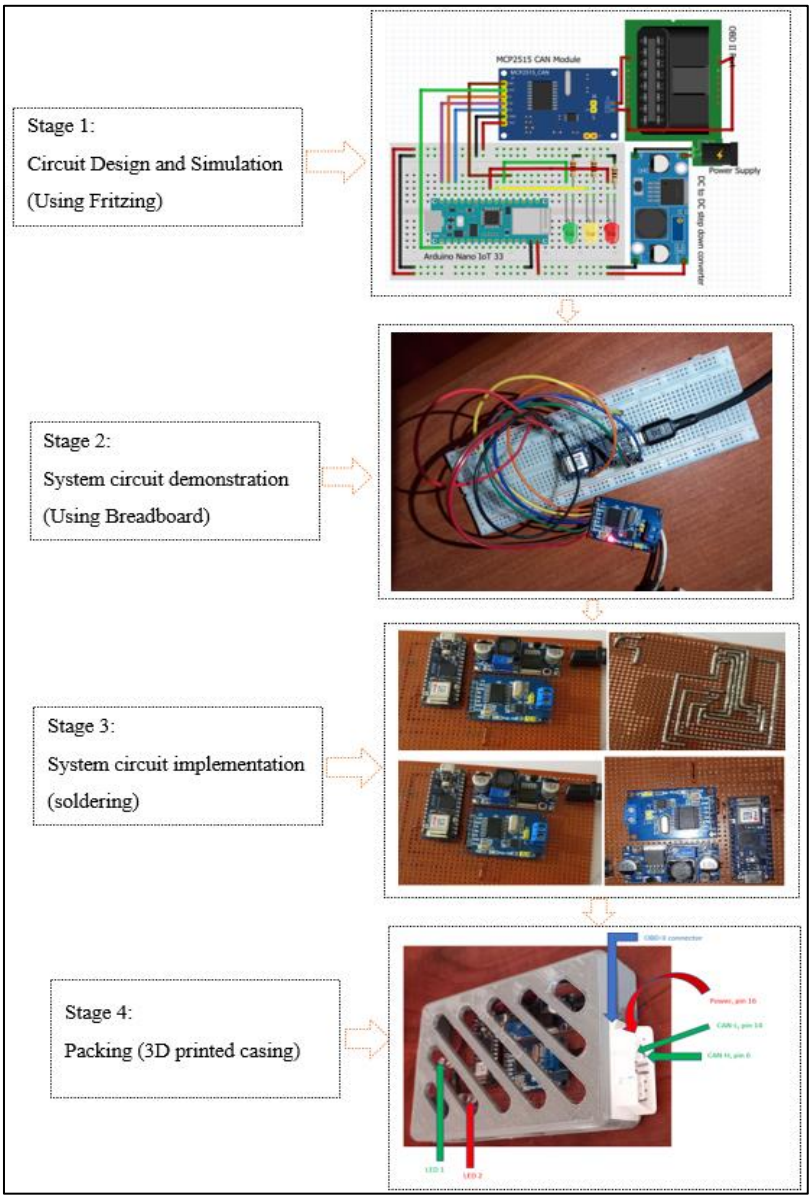


Figure 10: Hardware design and development stages

Stage 1 involved circuit design and simulation of the hardware components using Fritzing. Fritzing is a free, open-source software program for designing electronic circuits, making PCB layouts, and creating project schematics. Fritzing offers several pre-made electronic components that may be dragged and dropped into a virtual breadboard and then joined to form a circuit. Furthermore, Fritzing supports the design of custom components and allows the export of the circuit design as an image or a PCB layout file for the development of the embedded device. The software is preferred to alternative software because of its ease of use,

availability of pre-made components, and flexibility to generate unique components. It was used to create a schematic diagram and PCB for the system, as shown in Fig. 11.

The MCP2515 (CAN module) and Arduino Nano 33 IoT demonstrate how data acquisition is carried out from the CAN network of the vehicle using CAN-H and CAN-L communication lines. Devices and microcontrollers can communicate with each other's applications through the Controller Area Network (CAN), a vehicle's CAN bus. Each vehicle's two electrical lines (CAN Low and CAN High) are utilized to send and receive ECU data (AutoPi, 2022). From Fig. 11, the left-hand side (Transmitter) shows the CAN port interface of the vehicle, while the right-hand side (Receiver) is an embedded system that retrieves data from the vehicle.

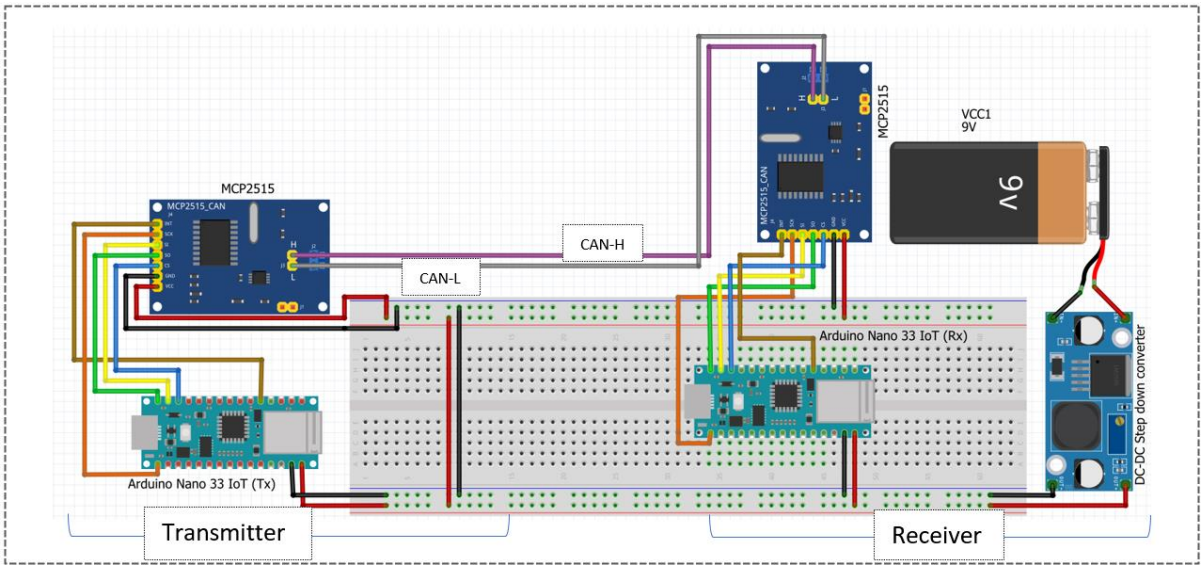


Figure 11: Fritzing breadboard drawing of the system

The MCP2515 CAN module can be interfaced with the Arduino Nano 33 IoT through the SPI communication protocol. The SPI pins of the MCP2515 module are connected to the corresponding SPI pins of the Arduino Nano 33 IoT. The CAN pins of the MCP2515 module are connected to the CAN bus of the vehicle. The Arduino Nano 33 IoT is programmed to communicate with the MCP2515 module and acquire data from the CAN bus. The acquired data can then be transmitted to a remote server using the built-in WiFi module of the Arduino Nano 33 IoT. The pin connections for the Arduino Nano 33 IoT (Microcontroller) and MCP2515 (CAN module) in Fig.11 are listed in Table 4 (CAN module).

Table 4: MCP2515 and Arduino Nano 33 IoT pinout connections

MCP2515	Arduino Nano 33 IoT
INT	D2
SCK	D13
SI	D11
SO	D12
CS	D10
GND	GND
VCC	VIN

Figure 12 illustrates the Fritzing diagram when the device is directly connected to the OBD 2 port of the vehicle. The vehicle supplies a voltage of 24V, and the DC-DC converter down-converts it to 5V to power the circuit board for the MCP2515 (CAN module) and the Microcontroller (Arduino Nano 33 IoT). OBD II port is the vehicle interface for diagnostic devices.

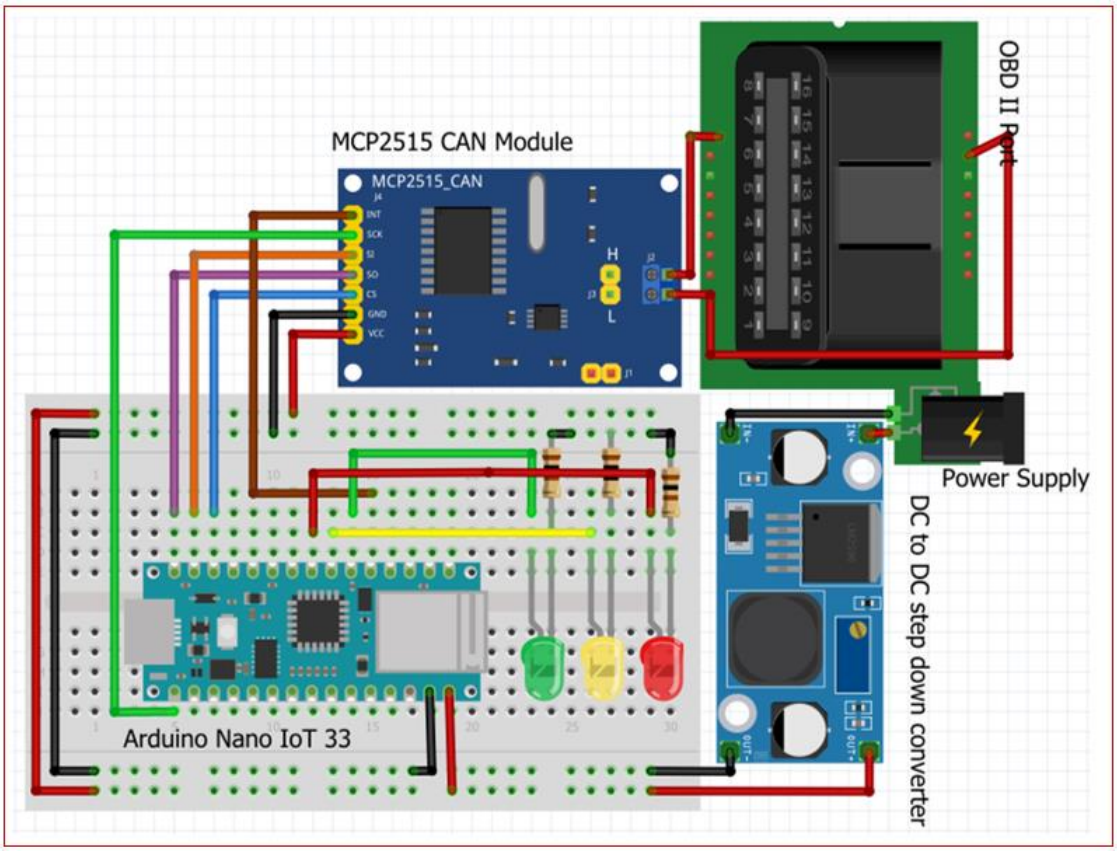


Figure 12: Fritzing receiver end of the system

Using Fritzing software (Fritzing, 2022), the schematic diagram circuit of MCP2515, Arduino Nano 33 IoT, and LM2596 (DC to DC step-down converter) in Fig. 13 was developed. The MCP2515 module was linked to the SPI pins of the Arduino Nano 33 IoT, which connects with the vehicle's CAN bus. The LM2596 step-down converter is connected to the VIN and GND pins of the Nano 33 IoT to convert the vehicle's 12V supply voltage to 5V, which is required to power the Nano 33 IoT. Connecting the power and ground pins of the MCP2515 module and LM2596 converter to the vehicle's 12V supply completes the circuit. This circuit interfaces the Arduino Nano 33 IoT with the MCP2515 module and receives it from the CAN network of the vehicle. This schematic system diagram made it easy to have a compact system design, easy testing and repair, and minimal errors when assembling the system parts.

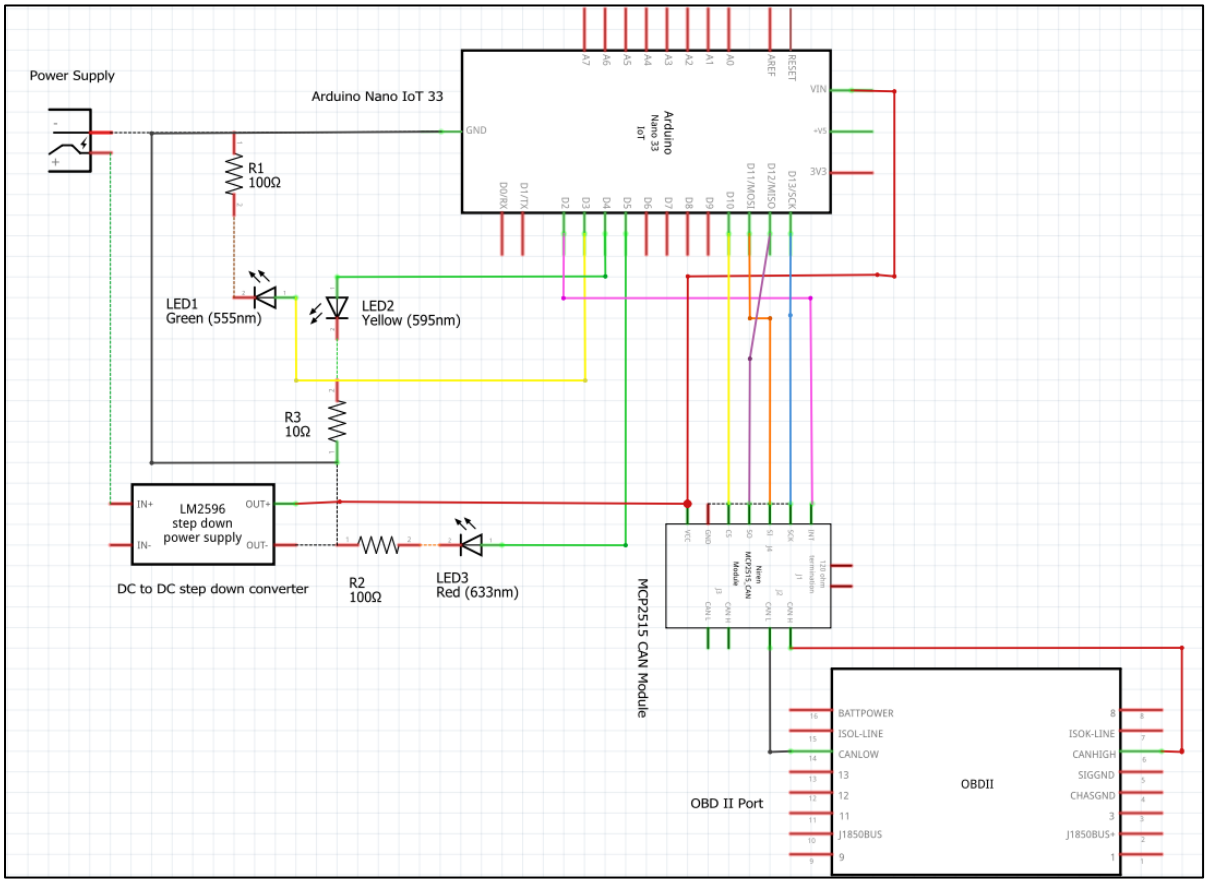


Figure 13: Hardware system schematic diagram

Stage 2 describes how the prototype's circuit simulation and programming were accomplished using a breadboard, Arduino IDE (Integrated Development Environment), and the developed circuit. Arduino IDE is a free, open-source software tool for programming and developing code for Arduino boards. This IDE offers a user-friendly interface for creating, developing, and uploading code to Arduino microcontrollers. Because of its easy user interface and

comprehensive documentation, the Arduino IDE is popular among novices. It supports various programming languages, including C and C++, making complicated programs simple to build. It also has a big developer community that has provided libraries and code snippets that may be readily integrated into projects. These qualities make it a popular choice for developing this research project's remote vehicle health monitoring system. With the help of the MCP2515 CAN module and this IDE, an Arduino Nano 33 IoT could extract data from a vehicle's CAN Bus and send it to the system cloud server. Figure 14 illustrates the offboard circuit of the developed system and how data is transmitted and retrieved over the CAN bus of the vehicles using the hardware components (circuit wiring diagram below) programmed in the Arduino IDE.

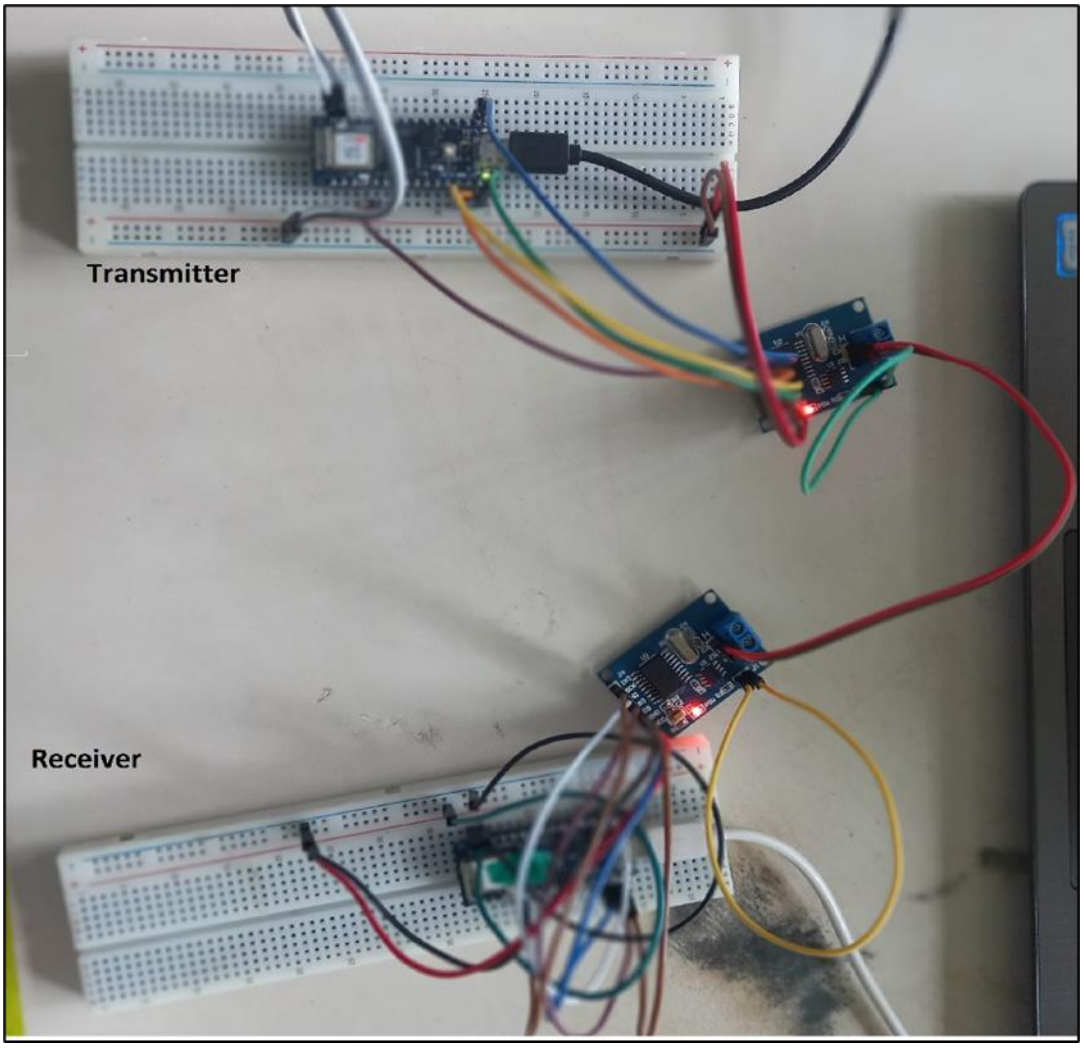


Figure 14: Hardware wired connection of breadboard

The output of the offboard circuit wiring after the simulation is illustrated in Figs. 15 and 16, one being the transmitter output and the other being the receiver.

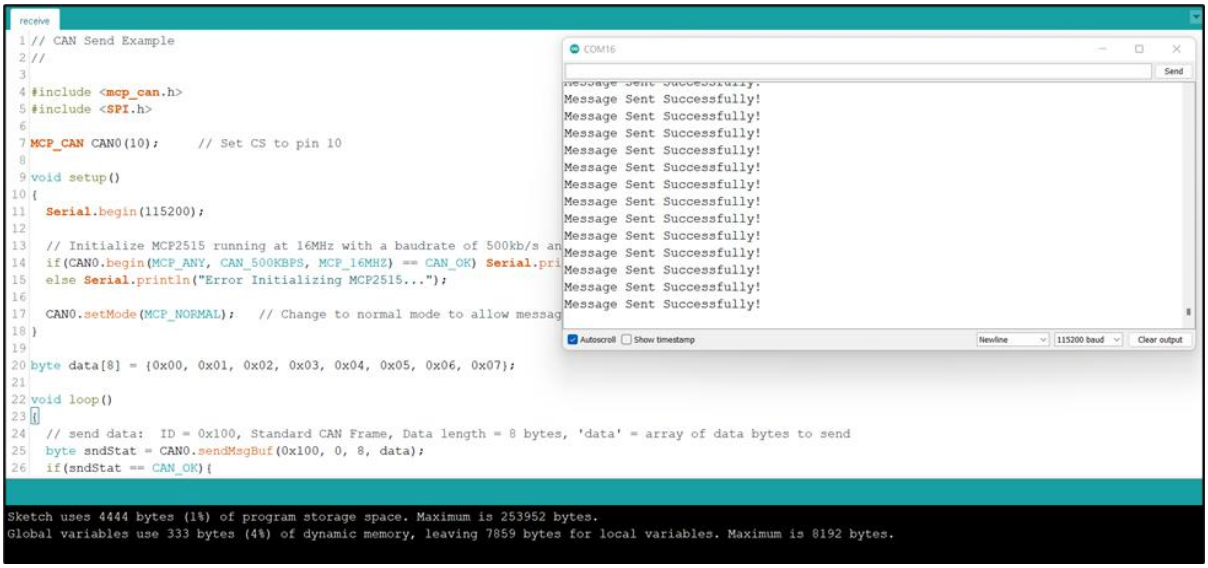


Figure 15: Transmitting data from the vehicle OBD-II

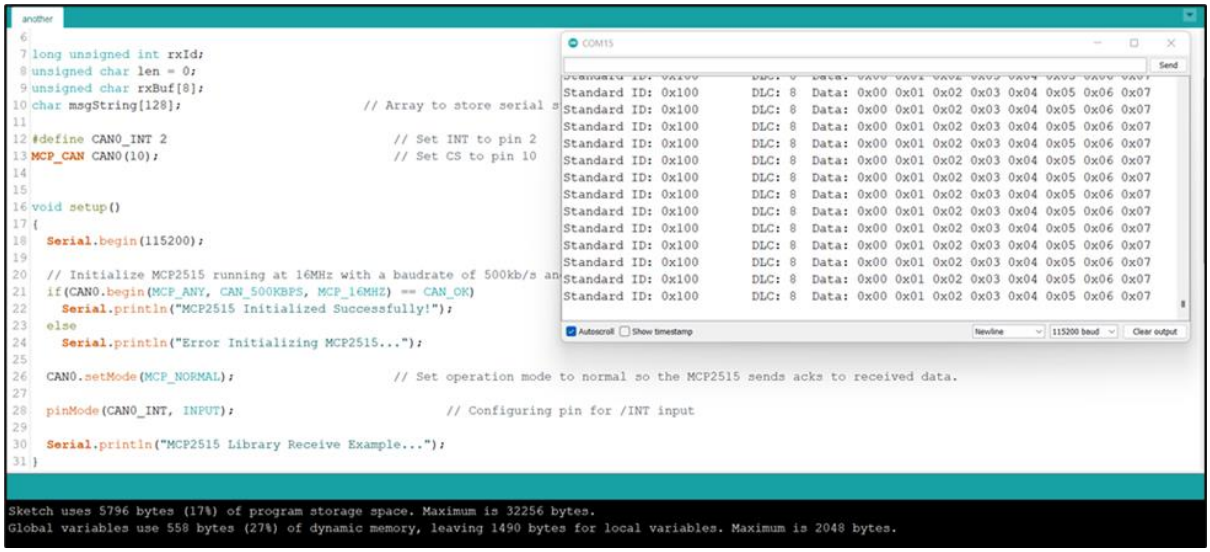


Figure 16: Data received from vehicle OBD-II

Stage 3 involves circuit design and integration by soldering, shown in Fig. 17 (front and back parts). The height of the circuit board with hardware components soldered on it is 14.8 mm/inch, the length is 118.7 mm/inch, and the width is 74.6 mm/inch. These measurements, shown in Fig. 18, were acquired with a vernier calliper to help determine the size of the casing to be printed with a 3D printer.

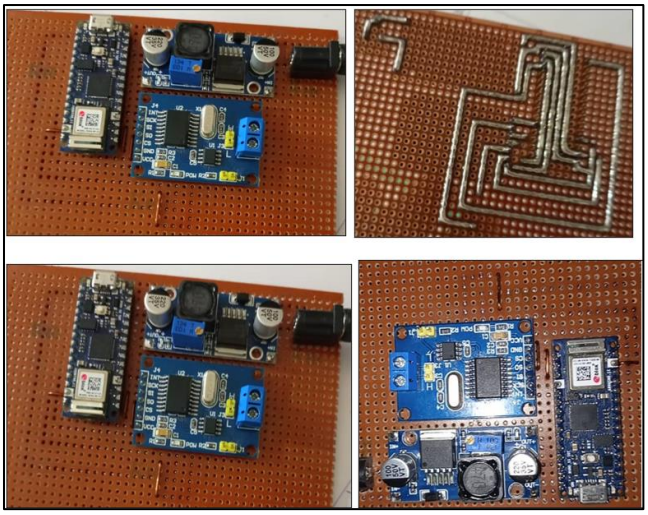


Figure 17: Soldered circuit of the system

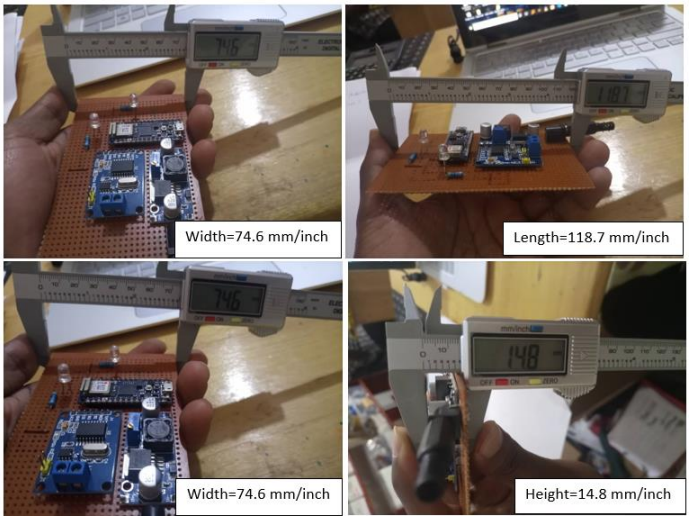


Figure 18: Circuit board measurements for 3D printing

Stage 4 explains how circuit board measurements in Fig. 18 were used to print out a 3D hardware casing of the developed embedded device using a creality ender 3D printer. The 3D printed casing has pinouts that match those of the CAN port in Kayoola electric vehicles where the main focus was put on power, CAN-H, and CAN-L pins which are communication lines in both CAN module in the embedded system and CAN bus in the Kayoola Electric Buses. The 3D printing, also referred to as three-dimensional printing or Rapid prototyping (RP), allows designers to create tangible prototypes of their designs quickly. For the case of the 3D Printed hardware casing in Fig. 19, the Creality Ender 3D printer, which is a desktop 3D printer that employs fused deposition modelling (FDM) technology to layer-by-layer manufacture 3D things using materials such as Polylactic Acid (PLA) and Acrylonitrile Butadiene Styrene (ABS) was used. It has a build capacity of 220 x 220 x 250mm with a resolution of 100 microns,

allowing it to produce complicated models and prototypes. Because of its low cost, ease of use, and ability to generate high-quality prints, the Ender 3D printer was preferred to other 3D printers for printing the casing of the embedded device. It was also very customizable and made simple to modify with various modifications and add-ons to the device casing.

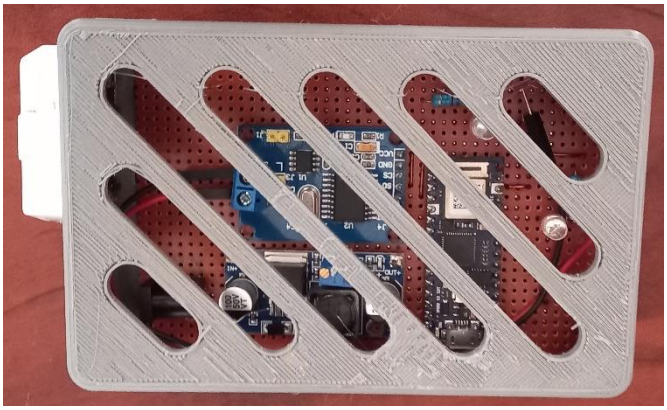


Figure 19: Hardware prototype

3.5.2 Software Development Requirements

This section comprises the programming languages, frameworks used in web application development, and architecture. Software design and development requirements are critical to developing a remote vehicle health monitoring system. These specifications define the features and function the system must have to monitor and assess vehicle health effectively. The specifications encompass various areas of the software development process, including hardware and software components, communication protocols, user interface, data processing, and storage. To guarantee that the system satisfies the desired aims and objectives, the system's design and development requirements must be properly defined and implemented.

(i) Programming Languages and Frameworks Used

Python is a high-level programming language that is open-source and widely used in software development, data analysis, and artificial intelligence. For example, it was chosen to create the remote vehicle health monitoring system for Kayoola electric vehicles due to its easy syntax and code readability, allowing for faster development and better system maintenance. In addition, Python includes an extensive library of pre-written code, which makes it easier to develop the system with less code. It is also cross-platform compatible which may be used on various operating systems. Furthermore, Python has a vibrant community with frequent upgrades and many online resources.

Vue.js is a JavaScript framework for creating user interfaces and single-page apps. It's small, fast, and simple to combine with other libraries or current projects (Trio, 2022). Vue.js's architecture is reactive and composable, enabling efficient and flexible development. It also has a big and supportive community and numerous plugins and tools for extending its functionality (Patel, 2021). Because of its simplicity, versatility, and performance, Vue.js is a popular choice for designing the web application for the remote vehicle health monitoring system for this research project.

Django Restful Framework (DRF) is a flexible and powerful toolkit for developing Web Application Programming Interfaces (APIs). It's built on the Django framework and offers tools to assist developers in designing both simple and maintained APIs (Quintagroup, 2022). DRF was chosen because it can handle complicated serialization, authentication, and URL routing, among other things. Furthermore, DRF supports many formats, including JSON and Extensible Markup Language (XML), making it an excellent choice for developing RESTful APIs that were used in developing the web application for the remote vehicle health monitoring system.

(ii) The Web Application Architecture

The languages used in web application development are VueJs, HTML, CSS, and Javascript for the front end, Django, and SQLite for the back end. The web application architecture used in its development is shown in Fig. 20.

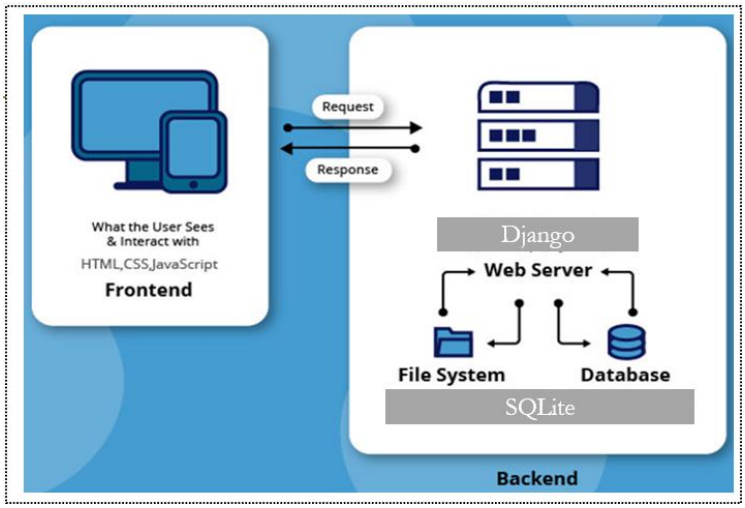


Figure 20: Web App Architecture

3.6 Methods

The methods section is a critical part of this dissertation research project. This section details the interviews and observation research methods, system development approach, data collection, data analysis methods, and system testing processes. The section also describes how the research project was implemented, including the software tools and technologies used in the study.

3.6.1 Interviews and Observation Research Methods

Qualitative methods of Interviews and observations were utilized throughout this research project development to understand exactly how the vehicles operate and how to address the problem.

The primary data sources were used, and seven KMC technical employees who work closely with Kayoola electric vehicles daily were interviewed face to face using a questionnaire attached in the appendices because of their expertise at the company and could provide significant information on the vehicles' systems. These employees included; four Drivers, two product support engineers, and one powertrain and Electrical and Electronic (EE) systems manager. During the interactions with different KMC employees, purposive sampling was used to select the seven employees engaged throughout this research project to develop the desired outcome. Also, several test experiments were carried out in the testing phase of the system so that the outcome is as proposed.

3.6.2 System Development Approach

An agile methodology is an iterative approach to software development that emphasizes client participation, adaptable planning, and delivering working software on time. It promotes collaboration, ongoing feedback, and adaptability to changing needs. The agile technique is preferred above other methodologies because it provides faster system delivery and the capacity to accommodate changing client needs. Extreme programming (XP) was desired in Agile methodologies since it takes the Agile development process to the extreme, focusing on the client's demands, performing iterative development, and introducing continuous integration. XP's four processes flow; Release Planning, Iteration, Acceptance Testing, and Small Release are illustrated in Fig. 21 (Digite, 2022). User stories and requirements are gathered before release planning to listen to customers and then satisfy their demands since

customer focus is one of the core values of KMC. System testing is later done for a better working system to win higher chances of user acceptance.

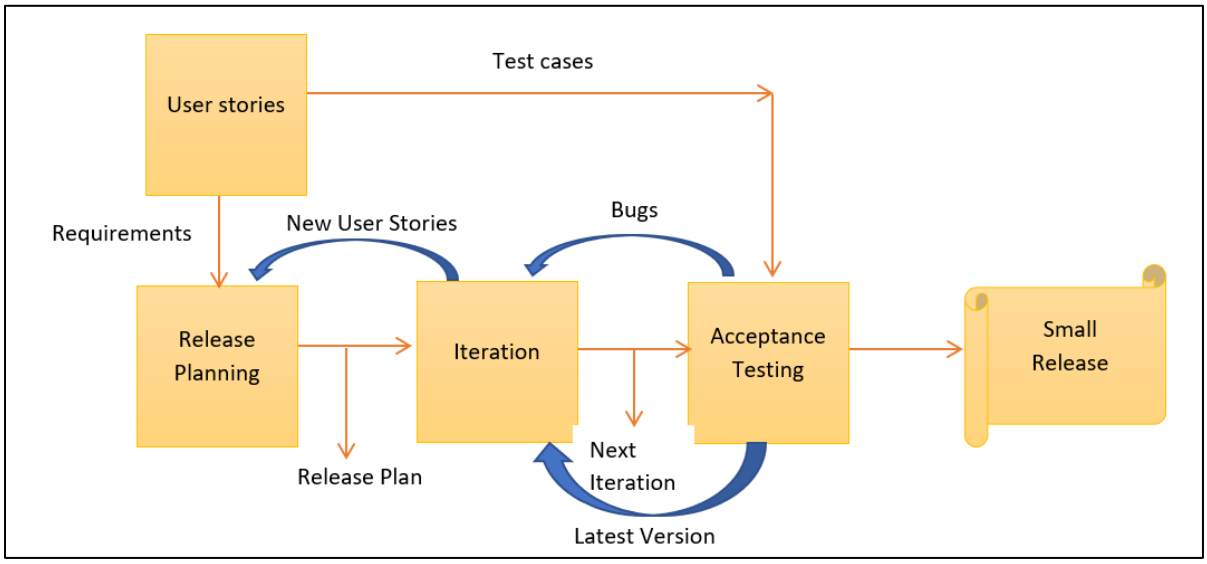


Figure 21: XP Programming Stages (Digite, 2022)

3.6.3 Data Collection Methods

A data-driven methodology built on real-time data with the help of the embedded device (prototype) that acquires data directly from the vehicle was used in this research project. The vehicle and prototype communication were established through the CAN communication network using CAN-H and CAN-L communication lines. The CAN-H and CAN-L lines are found at pins 6 and 14, respectively, for both the vehicle's On-Board Diagnostic (OBD) II port and the prototype's interface. This real-time data of five critical vehicle parameters of pack voltage, motor temperature, motor speed, pack current, and vehicle location gathered from different parts of the vehicle over the CAN communication network was later reviewed, analyzed, and monitored to derive a vehicle's health status and schedule maintenance. The developed embedded device for this research project was integrated into the vehicle through the CAN-H and CAN-L communication lines of the MCP2515 CAN module. The developed embedded communication system served as a link between the Vehicles and the web server.

3.6.4 Wireless Data Collection

An embedded system was developed specifically for this research project which was inserted in the vehicles' OBD (On board diagnostic) port, and data were wirelessly streamed through the CAN-H and CAN-L communication lines of the CAN bus module. The developed

embedded communication system served as a link between the Vehicles and the web server. The few parts of the EVs sampled for health monitoring are; the Battery Monitoring System (BMS), Vehicle Control Unit (VCU), Motor Controller, and DC-DC converter. Faults (diagnostic data) are shown as error codes, cross-referenced in the web application database to locate the correct problem, schedule repairs, or replace parts.

3.6.5 Controller Area Network (CAN) Bus Communication System

The Controller Area Network (CAN) bus is the communication network of the vehicle, which consists of the electrical wires CAN Low, and CAN High is a vehicle bus created to enable devices and microcontrollers to communicate with each other's applications. The CAN bus is also designed to manage heavy loads because of its five advantages.

The CAN network is the foundation of J1939. Mobile hydraulic systems and other heavy-duty vehicles, such as trucks and buses, follow the J1939 specifications of the Society of Automotive Engineers (SAE). The SAE J1939 standard outlines the CAN bus communication protocol used by heavy-duty automotive ECUs. Most current automotive use the CAN for ECU connection. CAN bus provides a "base" for communication; it does not provide a "language" for speech. Most heavy-duty vehicles adhere to the SAE J1939 standard, which the SAE created. In addition, J1939 provides a higher layer protocol (HLP) built upon the "physical layer" of CAN.

A set of specifications called SAE J1939 describes how ECUs communicate with one another in heavy-duty automobiles using the CAN Bus. J1939 employs a 29-bit identity and provides a higher-layer protocol. The 29-bit identifier used by the J1939 is described in the CAN 2.0B standard. The J1939 standard also provides a higher layer protocol (HLP) based on the CAN physical layer (AutoPi 2022). J1939 uses the CAN 2.0B protocol's 29-bit identity. A message intended for broadcast uses the identifier slightly differently from a message with a destination address ("PDU 1"). ("PDU 2").

Major characteristics of J1939 include the following: Kayoola buses are heavy-duty vehicles, so J1939 was developed by the Society of Automotive Engineers (SAE); speed is typically 250 kbit/s or 500 kbit/s; higher-layer protocol built on CAN; uses shielded twisted pair wire; J1939 messages are identified by 18-bit parameter group numbers (PGN), and J1939 signals are referred to as suspect parameter numbers (SPN).

3.7 Data Analysis Methods

The data analysis methods section is a crucial part of any research project that involves analyzing data and interpreting the live-streamed CAN data that comes in a complex format in a human-readable format. This section describes the data analysis methods used in the research, including the software and tools employed. The data analysis methods section is vital in determining the reliability and validity of the research results. It provides a clear picture of the data collected and analyzed, the methods used, and the results obtained.

3.7.1 Data Analysis and Processing

The acquired data from the CAN Bus using the Arduino Nano 33 IoT and MCP2515 CAN module comes in a complex format and unordered way, which needs to be sorted. Data pre-processing is the mining technique that transforms raw datasets into a readable and understandable format using Python programming. Processed the data acquired from Kayoola electric vehicles for this research project; it involved four major steps: data cleaning, data integration, data reduction, and data transformation, as shown in Fig. 22, to ensure data accuracy and consistency. The first step was data cleaning, which involved filling in missing numbers, reducing noise, locating or eliminating outliers, and resolving contradictions due to the unsorted nature of the data acquired from the vehicle. The next step was data integration, which combined data acquired from the vehicle into a single database. Data reduction followed, providing a compressed representation of the data set with significantly fewer data while producing comparable analytical findings. Finally, data transformation was necessary to convert the complex data format into a human-readable format by normalizing, aggregating, and transforming it. All these steps were accomplished after transmitting data wirelessly to the backend server to ensure the quality and reliability of the data used in the remote vehicle health monitoring system.

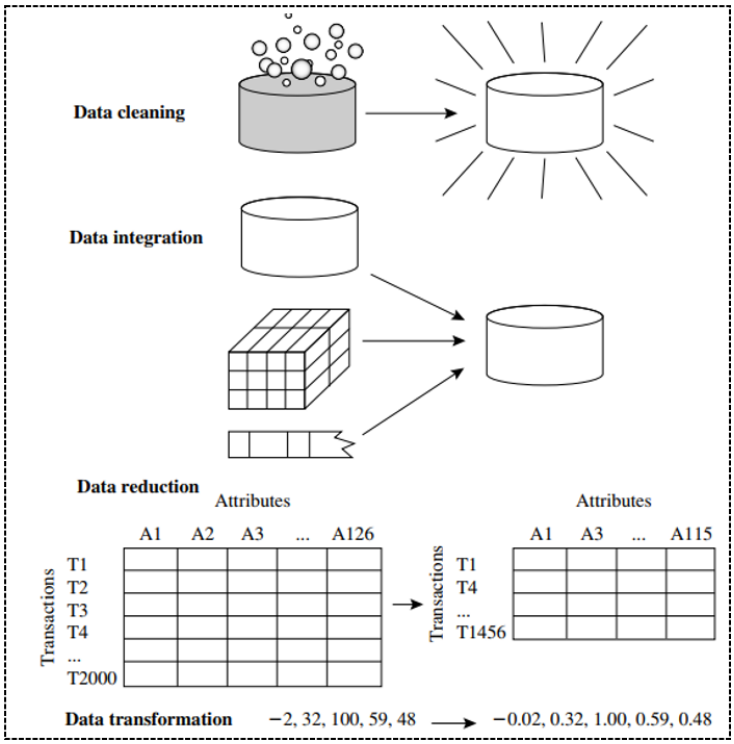


Figure 22: Data pre-processing steps (Han *et al.*, 2012)

3.7.2 Algorithm Formula

This formula illustrated in Fig. 23 was used to transform the pre-processed data into a human-readable format. It was implemented in the system's backend using Python for remote vehicle health monitoring on the dashboard. All the codes used are attached in the appendices.

Output = D x R+O where;

D = Decimal data

R = Resolution

O = Offset

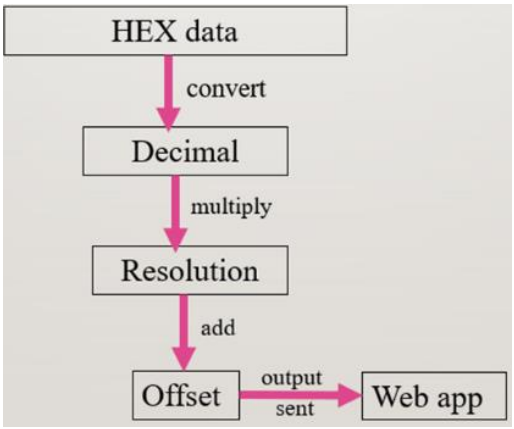


Figure 23: Illustration of the algorithm for data translation

The raw dataset received from the vehicle's CAN communication network has 298 265 data points transmitted in a CSV file from the Arduino IDE to the web app's backend for processing. However, this dataset keeps being updated after 30 seconds in a loop, and only five parameters were focused on because of the limited availability of resources from Original Equipment Manufacturers (OEMs). Therefore, the transmitted data has the following format in Fig. 24 before processing and formatting. It has eight columns of different data, but most importantly, the time stamp, which shows the time when data was generated; the frame ID, which shows different components in the vehicle identified by the source ID; and Data (HEX) which illustrates the status of different parts in the vehicle.

	Index	Direction	Time Stamp	Frame ID	Format	Type	Data Length	Data(HEX)
0	0	Receive	11:34:08.638.0	0x1861a6f3	Data	Extend	0x08	d1 0c d0 0c d1 0c d2 0c
1	1	Receive	11:34:08.638.0	0x1862a6f3	Data	Extend	0x08	d2 0c d1 0c d3 0c d3 0c
2	2	Receive	11:34:08.638.0	0x1819a6f3	Data	Extend	0x08	3d 3d 3d 3d 3d 3d 3d 3e
3	3	Receive	11:34:08.638.0	0x1863a6f3	Data	Extend	0x08	d3 0c d3 0c d2 0c d3 0c
4	4	Receive	11:34:08.638.0	0x1864a6f3	Data	Extend	0x08	d4 0c d3 0c d3 0c d4 0c
...	...	...	...	...	...	...	...	...
298260	298260	Receive	11:41:44.088.0	0x1882a6f3	Data	Extend	0x08	ce 0c ce 0c ce 0c ce 0c
298261	298261	Receive	11:41:44.088.0	0x1883a6f3	Data	Extend	0x08	cd 0c ce 0c ce 0c ce 0c
298262	298262	Receive	11:41:44.088.0	0x18f0010b	Data	Extend	0x08	cf ff 30 ff ff 0d ff ff
298263	298263	Receive	11:41:44.088.0	0x18febf0b	Data	Extend	0x08	f0 0a 7d 7c 7c 7b ff ff
298264	298264	Receive	11:41:44.088.0	0x1884a6f3	Data	Extend	0x08	d0 0c ce 0c ce 0c cf 0c

298265 rows × 8 columns

Figure 24: Raw dataset format

3.8 System Testing Processes

This section explains the system operations and functionalities, starting with the individual systems and then the entire system. The system went through several testing stages for its full functionality; unit tests, integration, system, functionality, acceptance tests, and performance. The hardware (embedded system) and the software went through these testing stages before the final integration of the whole system (Technologies, 2022).

3.8.1 Functional Testing Stages

Unit, integration, system, and acceptance testing in Fig. 25 are all part of the functionality stage. Functional tests were done to verify all the requirements of the proposed system. The tests done

here were only focused on verifying the output of an action. The entire system's functional operations were tested in this phase to determine how efficiently they run together.

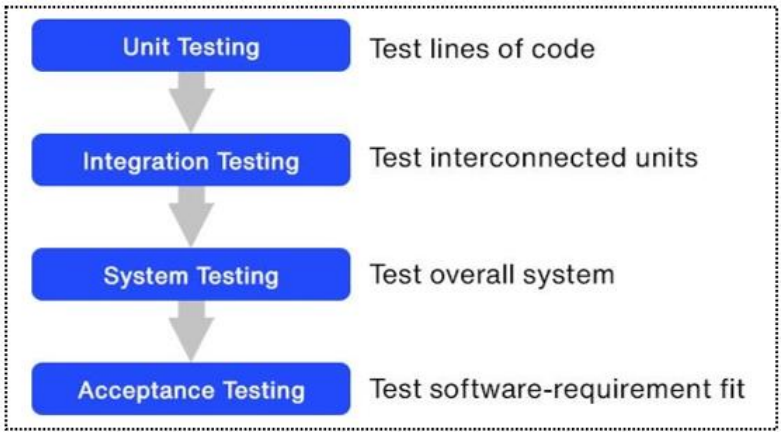


Figure 25: System testing stages

3.8.2 Unit Tests

This phase has unit functional tests, individual unit program tests, and unit procedural tests. These are initial functional tests for individual hardware components and system software modules. These were done to ensure the hardware and software functionalities were as anticipated.

3.8.3 Integration Tests

These helped verify that different system components and modules can work well together. This involved hardware integration separately as well as integration of software modules alone. Individual system integration was followed by the entire system integration, where the hardware was linked to the software. This combines all the programs and modules within the system, which are later tested at once.

3.8.4 System Testing

System testing is the first level of testing done on the entire application. Assessing if the system has complied with all of the specified requirements is the objective at this level. This system testing phase assisted in determining whether the overall system functioning complies with the objectives' technical, functional, and other requirements. Its primary goal is to assess the complete system specs.

3.8.5 Acceptance Testing

Acceptance tests are the official evaluations to confirm that a system complies with user requirements. These were completed while the system's entire web application was already operational. The entire system's operations that didn't match the established goals and weren't as necessary were deleted. This round of system testing for the created remote centralized vehicle health monitoring system was likewise regarded as the last. The user tested the system in this last stage to see if it addressed and resolved the issue (Technologies, 2022).

3.8.6 Non-Functional Testing Stages

This stage involves security, performance, Usability, and compatibility system testing. Performance tests assess how a system performs under specific workloads and environmental circumstances. These evaluations aid in gauging the system's dependability, speed, scalability, and responsiveness. Usability testing looks at an application's appearance, feel, and user-friendliness from the user's perspective. Security testing is performed on software, applications, and websites to determine how well they are protected against internal and/or external threats. This testing examines how well the software is protected from viruses and malicious software and the security and robustness of authorization and authentication procedures. It also looks at how the software responds to hacker attacks and malicious software and how software is maintained for data security after an attack. This type of testing validates how the software works and behaves in various environments, such as hardware, web servers, and network environments. Finally, the software is tested for compatibility with various configurations, databases, browsers, and their newest iterations. Compatibility testing is done by the testing team (Softwaretestinghelp, 2022).

Table 5: Summary table of the tests done

S/N	Requirement	Description	Status
1	Unit tests of hardware components	The Microcontroller and CAN bus module were tested individually for their functionalities.	Passed as expected
2	Hardware integration	All the components were later integrated to check whether they worked well together.	Passed as expected
3	Transmission of data between components	This was demonstrated using the left-hand side (Transmitter) as the CAN port interface of the vehicle while the right-hand side (Receiver) is an embedded system that acquires data from the vehicle in Fig 3.26.	Transmission and reception successfully
4	WiFi connectivity	The Microcontroller was tested to check whether it could connect to WiFi since the means of data transfer in the system was wireless.	Passed successfully

3.8.7 Embedded System Unit and Integrated Tests Processes

The components were tested individually, and the unit tests passed as expected; the embedded device was later integrated into the Kiira EV to test its functionality. MCP2515 and Arduino Nano 33 IoT (Microcontroller) were first tested together to check whether they function properly, as shown in Fig. 26. The functionality test was done on a breadboard with one set of components acting as the transmitter and the other as the receiver. This functionality test passed as expected.

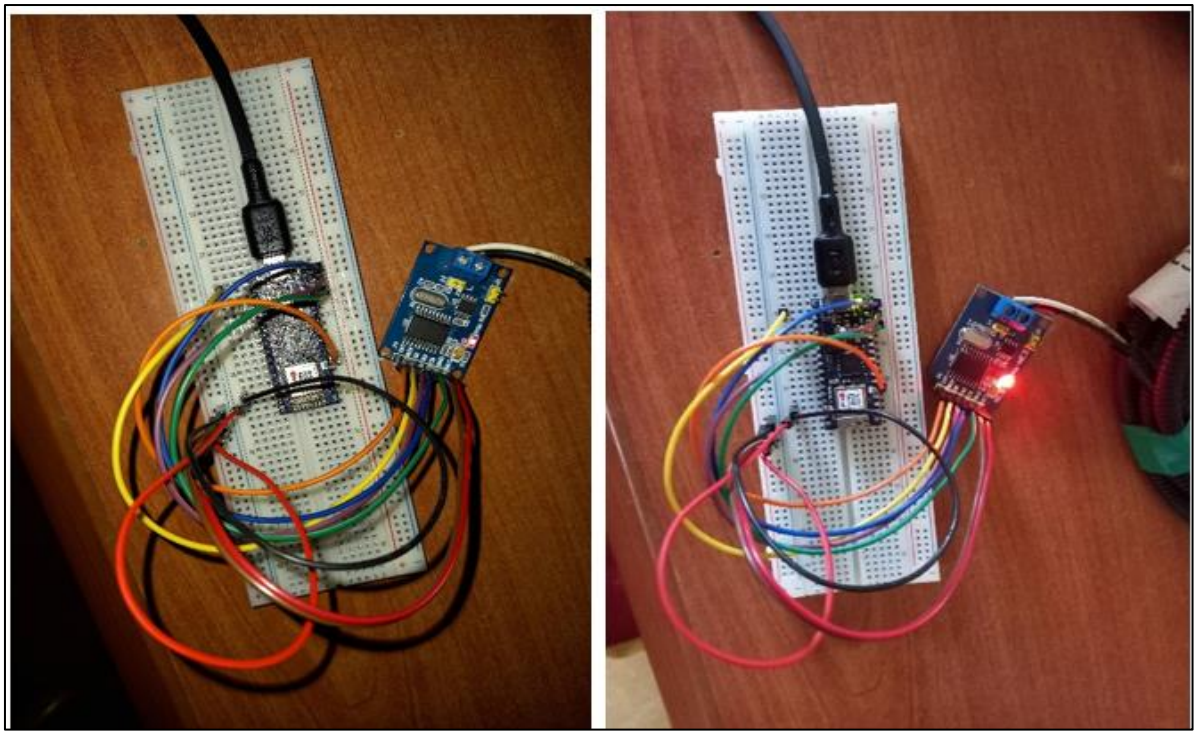


Figure 26: Communication between the MCP2515 and the Microcontroller

After the functionality testing of the components, the transmitter and the receiver were integrated, as shown in Fig. 27. One side with Microcontroller and CAN module acted as the transmitter (vehicle OBD-II port), and the other side with the same components acted as the receiving end. The functionality tests of the components after integration also passed as expected (Fig. 28).

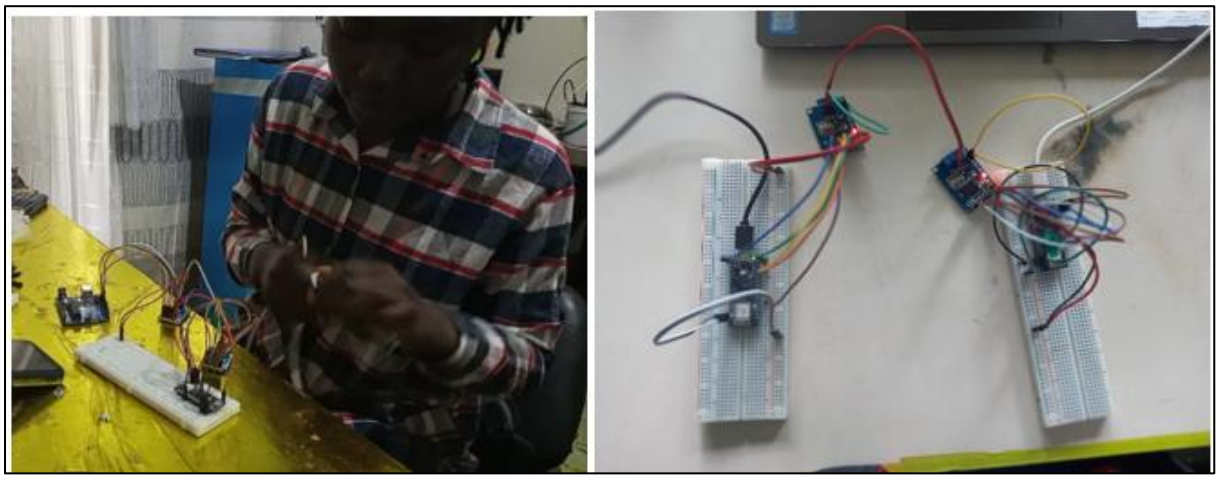


Figure 27: Hardware transmitter and receiver integration

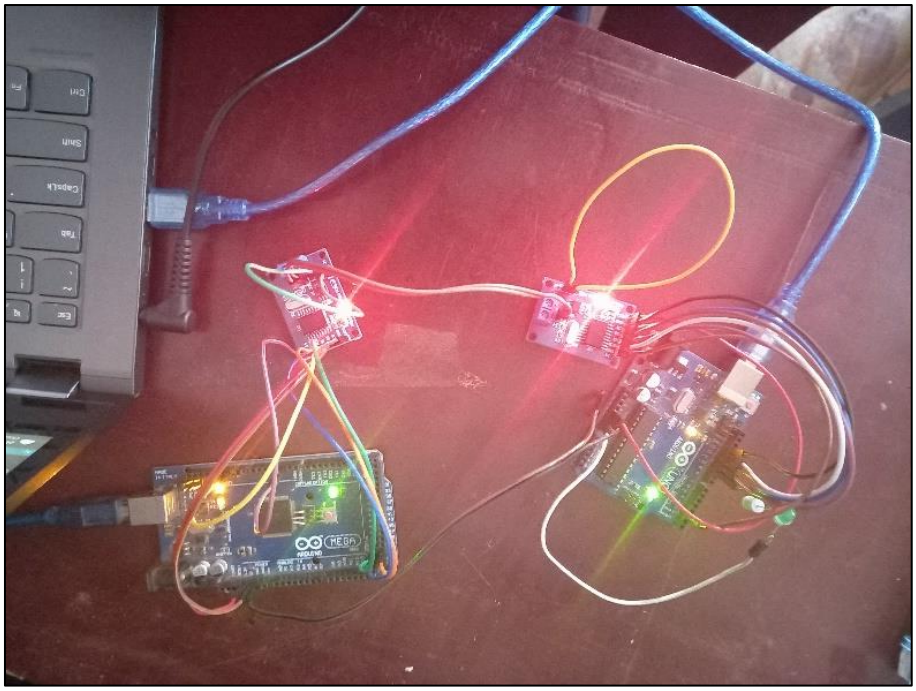


Figure 28: Functionality tests

The embedded device was tested offboard and then integrated into the Kiira EV to acquire data from its CAN communication network. The Arduino IDE was used for testing, providing a text editor for coding, a message box, a console, and a toolbar. The IDE also enabled easy uploading and communication with the Arduino hardware. Results showed that the MCP2515 CAN module and Microcontroller could receive data from the vehicle's network. Figure 29 displays the output of components, demonstrating the successful retrieval of CAN messages from the Kiira EV. Testing was carried out on the vehicle, with Fig. 29 as an illustration of raw data streaming from different vehicle subsystems. This data was sorted and translated from the web application's backend to remotely monitor the vehicle's status.

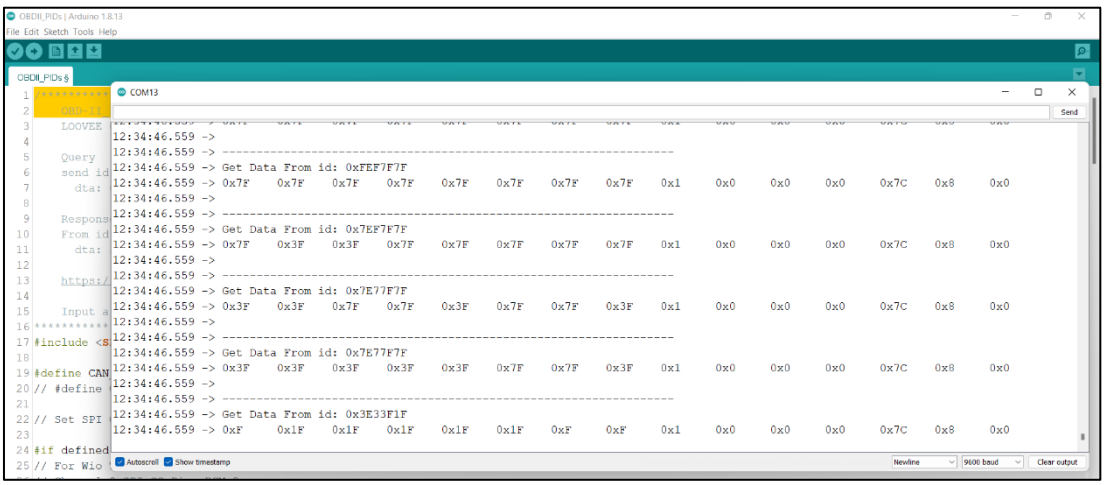


Figure 29: Receiver results of the hardware prototype

The received data from the Kiira EV's CAN communications network must be transmitted wirelessly to the system's database, which calls for Internet of Things (IoT) technology integration. The WiFi connectivity and functionality underwent testing and met expectations, as illustrated in Fig. 30.

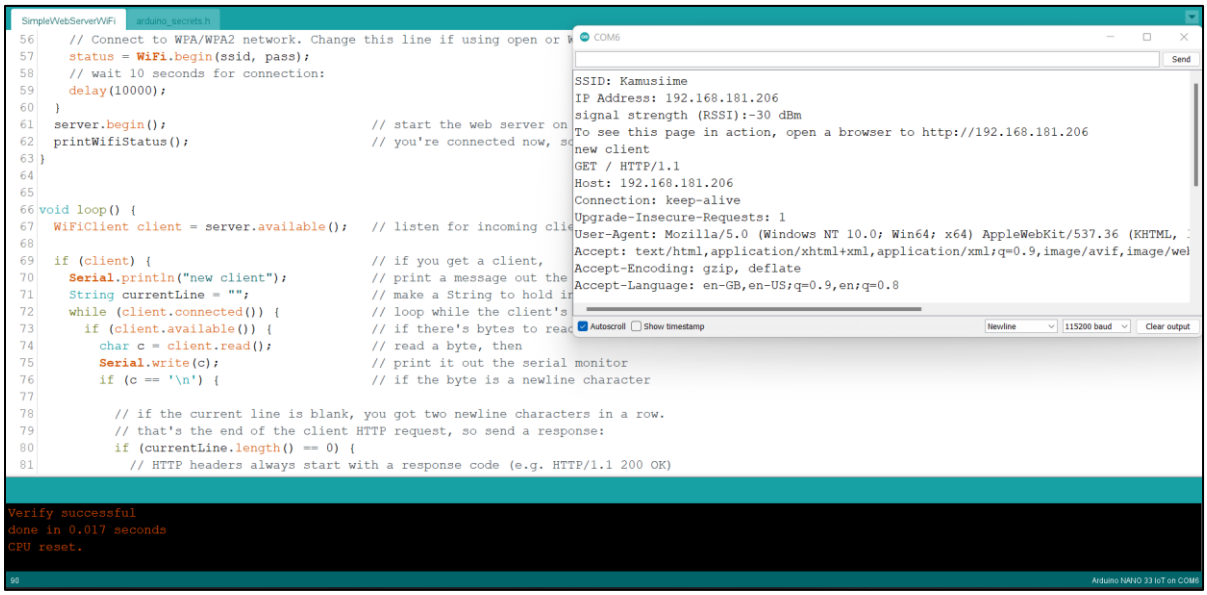


Figure 30: WiFi connectivity results of the hardware

3.9 System Validation

The system was demonstrated to KMC technical employees in the presentation, where a poster presentation is attached in Appendix 9. This was followed by a user testing questionnaire from respondents who attended the prototype demonstration presentation. The results of System validation in Fig. 31 show that 78.6% were males and 21.4% were females. Also Figs. 32, 33 and 34 illustrate the validation results from the questionnaire, and the remaining parts of the validation questionnaire are in Appendix 4.

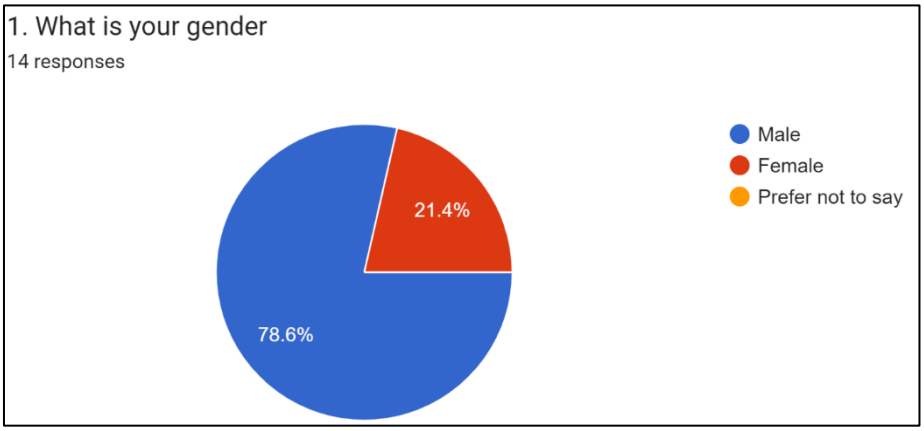


Figure 31: Validation result 1

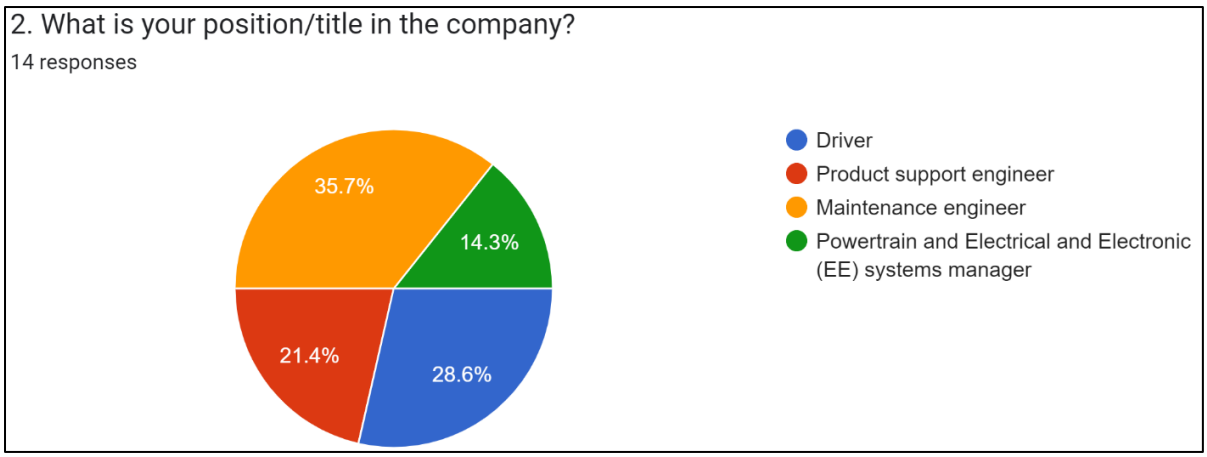


Figure 32: Validation result 2

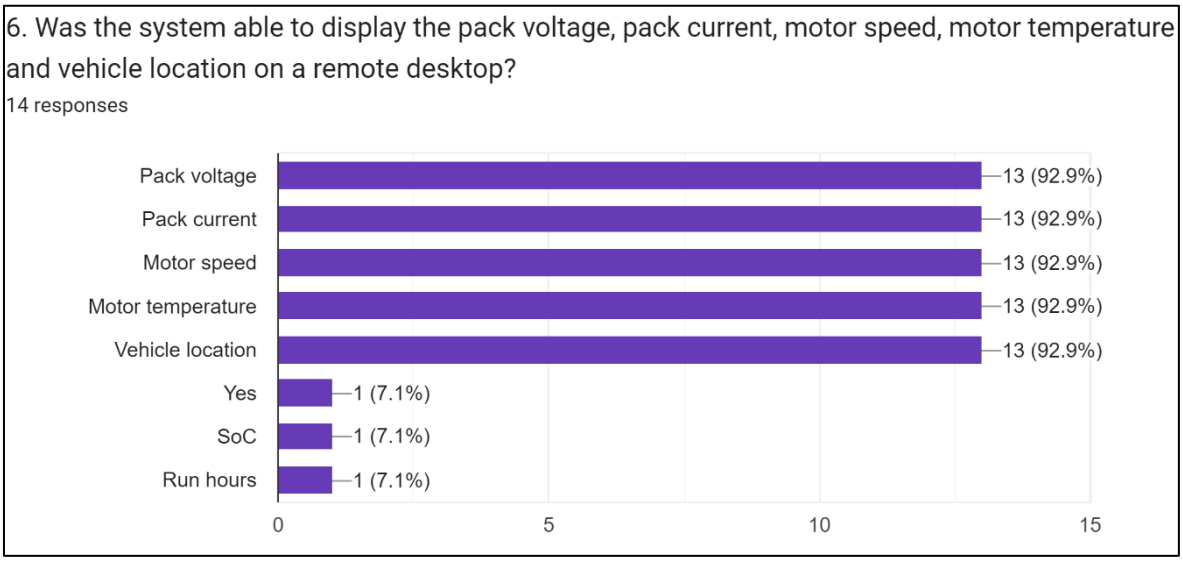


Figure 33: Validation result 3

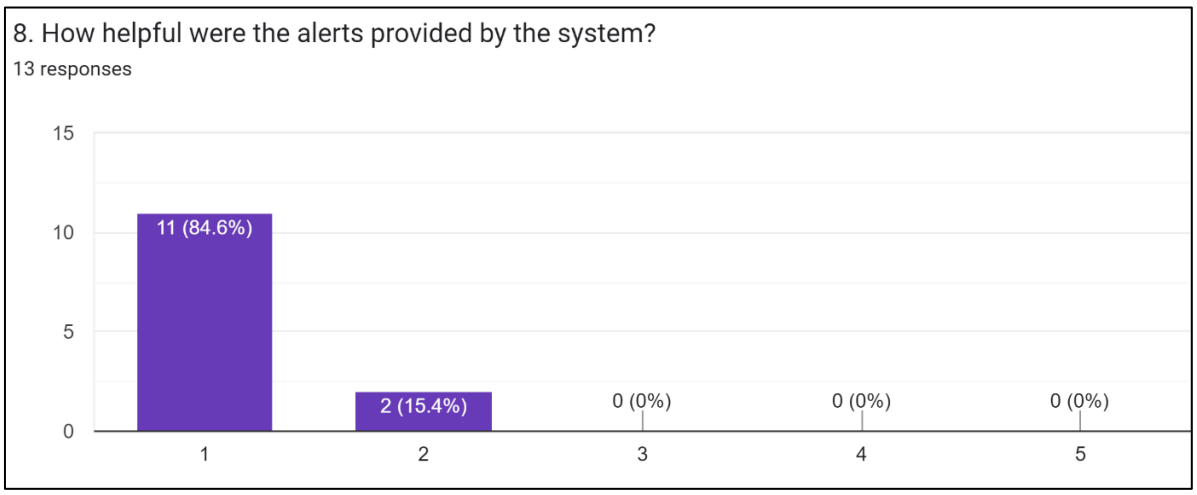


Figure 34: Validation result 4

CHAPTER FOUR

RESULTS AND DISCUSSION

4.1 Results

4.1.1 Hardware Prototype

The hardware prototype (embedded device) shown in Fig. 35, which can receive and transmit data wirelessly to the web application for remote monitoring, is one of the output results for this research project; it is an embedded standalone system with CAN-H and CAN-L communication lines that are identical to those found on the CAN port of Kiira EV. Table 6 details the specifications of the developed embedded system in Fig. 35.

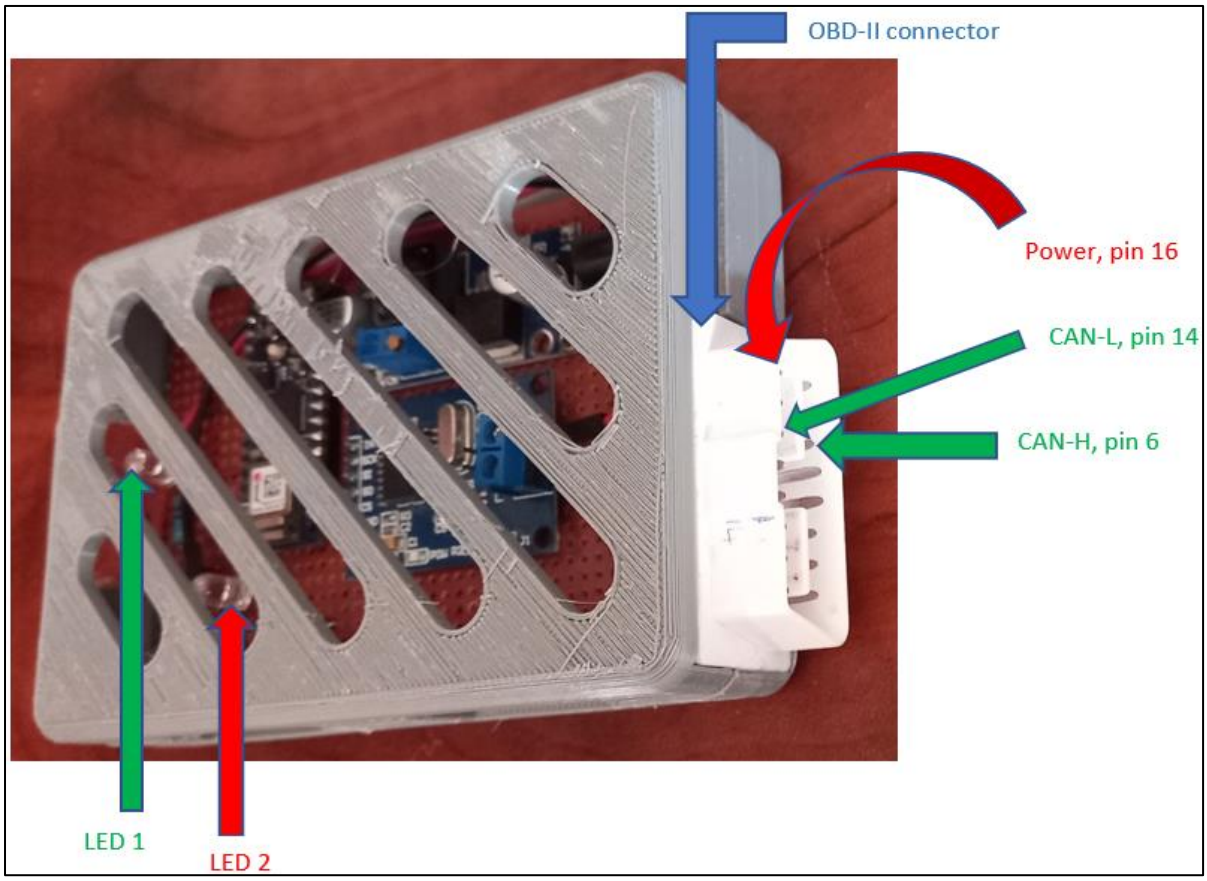


Figure 35: Hardware prototype with the required specifications

Table 6: Hardware prototype specifications

S/N	Part	Description
1	Arduino Nano 33 IoT	This is the main Microcontroller of the entire system. It was chosen because of its small size and having WiFi module integrated with it.
2	MCP2515	It's the CAN module with CAN-H and CAN-L communication lines
3	LEDs	It has 2 LEDs that light when powered and during data acquisition
4	DC-DC step-down converter	It converts high voltage from the vehicle through the power line at pin 16 to 5 volts that supply the system.
5	Power Lines	It is used to power the embedded system
6	CAN-L and CAN-L	These are communication lines that match the CAN bus communication lines

4.1.2 Web Application

A web application is the second and last output of this research project. It has a backend that can decode the received raw data into a human-readable format and display the results on its dashboard to complete the remote vehicle health monitoring system. The programming languages used throughout web app development are; the Vue.js framework for the front-end and Django restful framework for the backend, which was already explained in Chapter 3 in the web application architecture. The web application displays real-time results of the five parameters of focus and the health status of the Kiira EV features. The tests were carried out after integrating the embedded device into the Kiira EV, thus customizing the web application for the Kiira EV.

(i) Authentication Page

This is the procedure of confirming a user's identification, ensuring the system identifies and recognises the user. The system prompts the user to log in first before accessing its interfaces. If the user is logging in for the first time, they are prompted to create an account first, and it isn't the first time; it requires the credentials used when creating the account (Fig. 36).

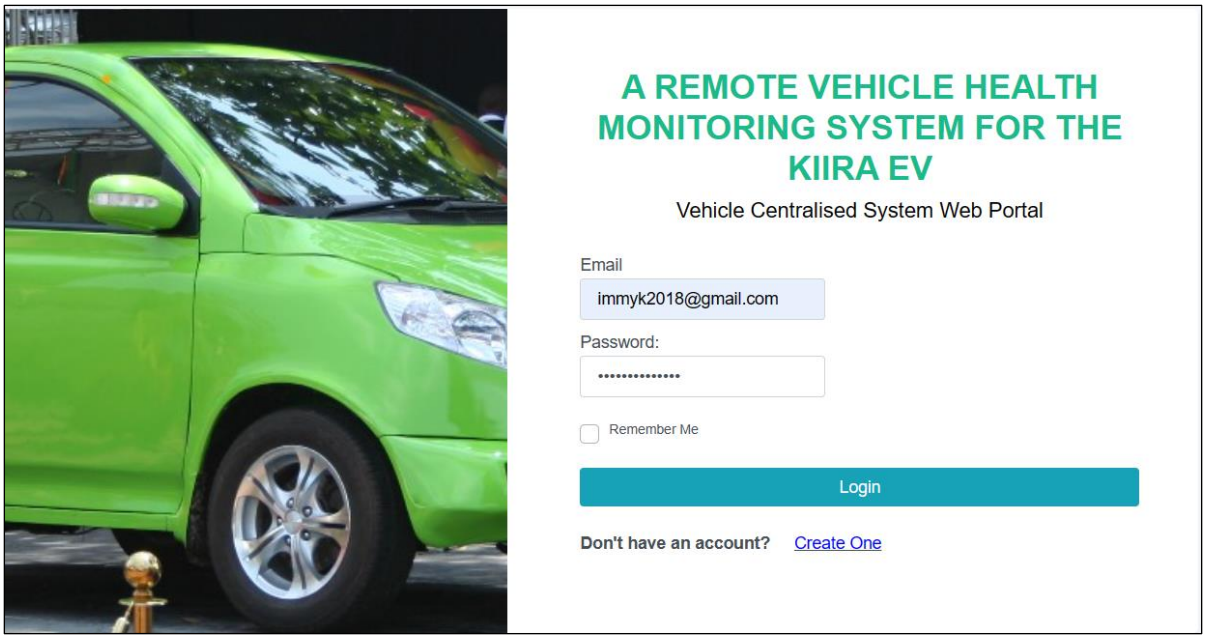


Figure 36: Authentication page

(ii) Dashboard Page

The web app dashboard shown in Fig. 37 is the first page after logging into the system that shows the vehicle highlights. It displays the total number of vehicles a company has to cater for space of testing the system on other vehicles, the Diagnostic Trouble Codes (DTC) for the faults that occurred in a week, the cleared and uncleared DTCs, the average vehicles' health status monthly, and the number of maintenance schedules planned per month. It also has a notifications bell that notifies the user of critical faults that need to be attended to urgently.

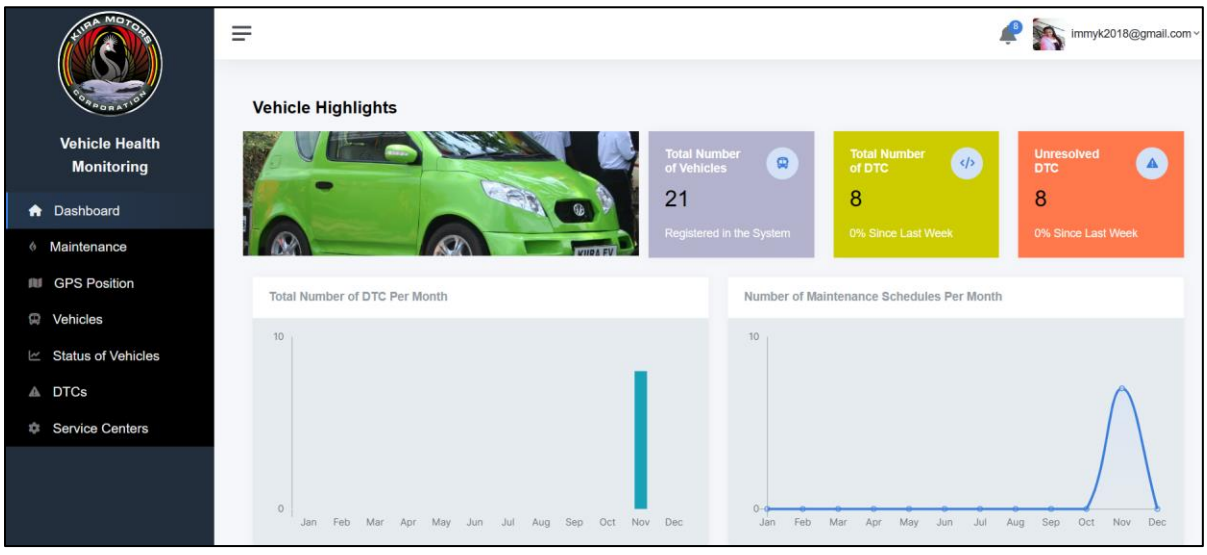


Figure 37: Dashboard page

(iii) Maintenance Schedules

The vehicle's health is supposed to be monitored daily to help the technical operators keep track of the vehicle’s status. The detected faults are benchmarked against the original status of the vehicle, where the results determine whether to inspect or replace the part depending on how critical the fault is Fig. 38 displays the automatically scheduled maintenance and planned maintenance schedules that are due. The user plans maintenance schedules based on the vehicle type and a brief description fault at hand; the system automatically categorizes the fault, whether urgent or not, using different parameters.

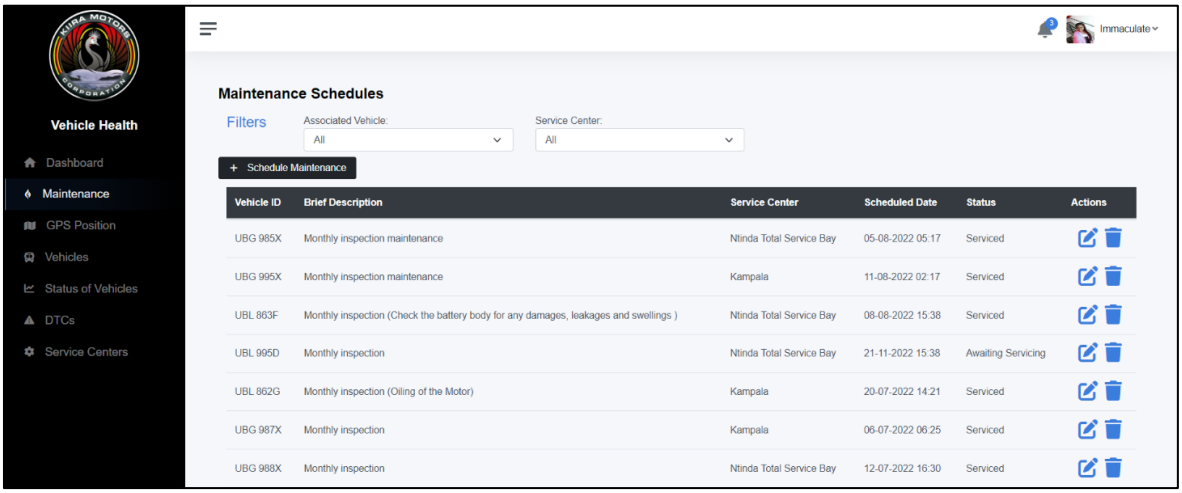


Figure 38: Maintenance schedules

(iv) Vehicles registered in the company

Figure 39 shows all the registered vehicles in the company. Once a vehicle is manufactured, its details, including the Identification Number (ID)/number plate, bus type, model, and year are entered into the system for proper monitoring and tracking.

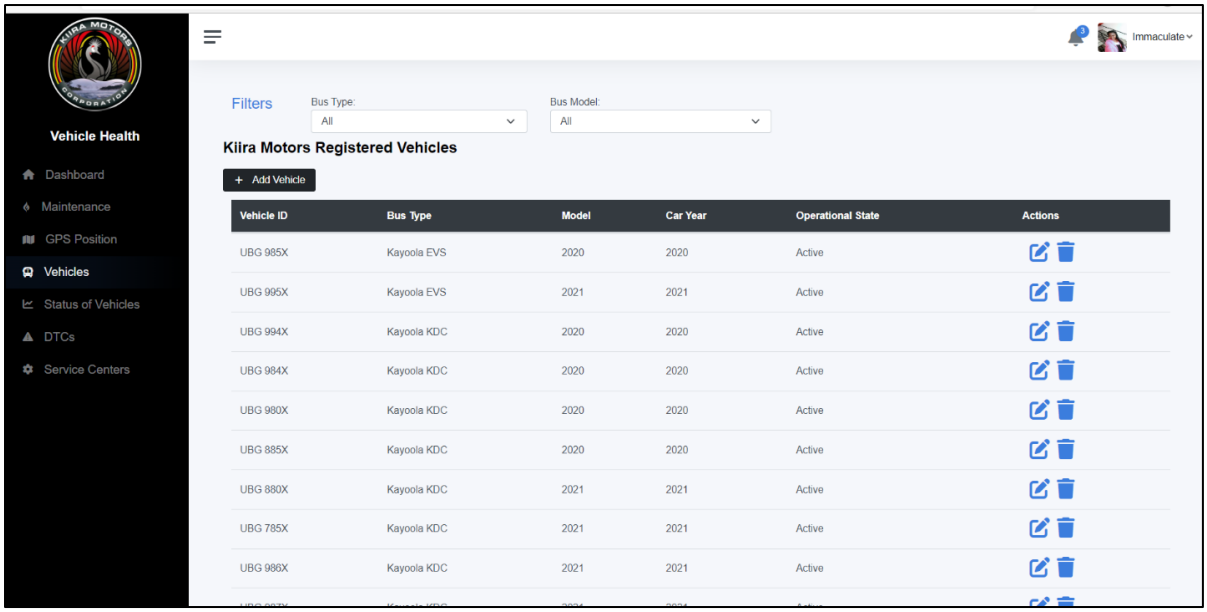


Figure 39: List of company-registered vehicles in the system

(v) Health Status of Vehicles

Figure 40 shows that before receiving data, the fields are empty. Figure 41 depicts the operational features of the remote vehicle monitoring system integrated into the Kiira EV, one of the concept electric vehicles made by KMC, which is the core system function of this research project. The system provides information on the health status of the vehicle. For instance, it indicates that the Kiira EV is active and displays real-time monitoring data as of 20-02-2023 at 4:46 PM, including five critical parameters of focus: pack voltage, pack current, motor speed, motor temperature, and the vehicle's location.

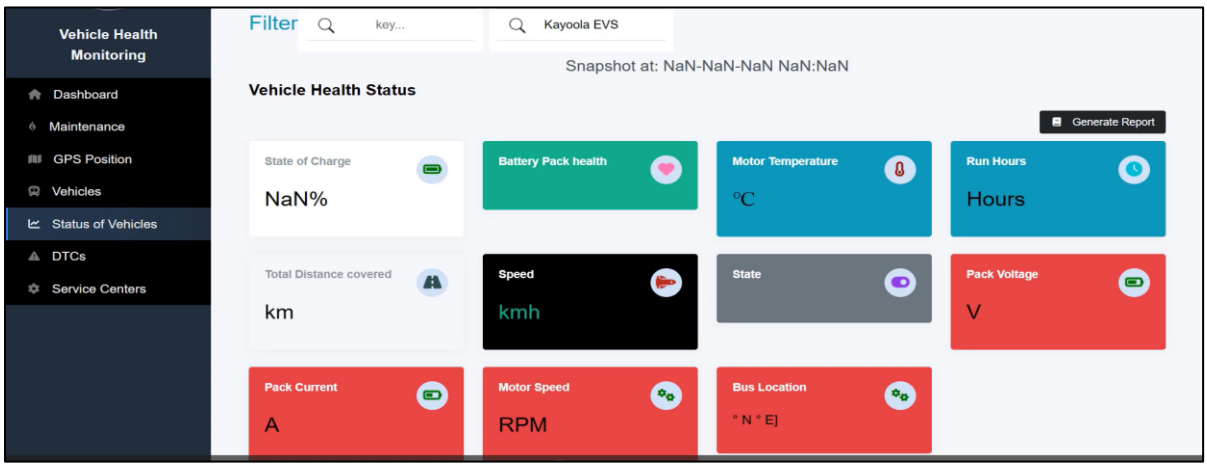


Figure 40: Health status page before receiving data

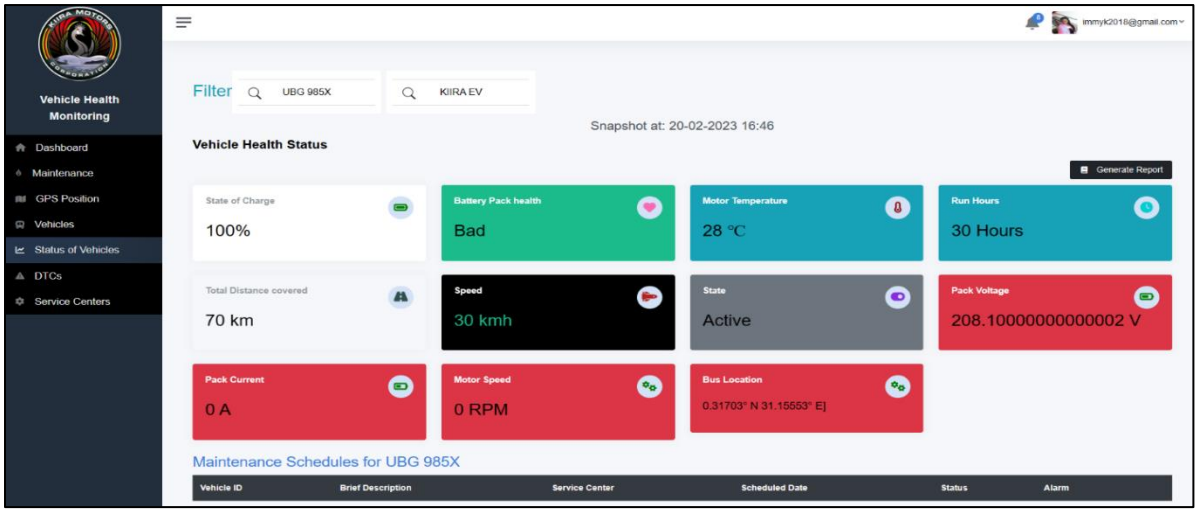


Figure 41: Kiira EV health status

(vi) Diagnostic Trouble Codes (DTCs)

A DTC code is a series of codes generated by a vehicle's system to alert the owner when a vehicle experiences a malfunction. The DTCs for Kayoola electric vehicles differ from those of other vehicles and have different meanings representing specific problems in the vehicle. The detected faults appear in the notification bell as a warning to help the owner identify and fix the critical issues within the vehicles. For the case of this system, as shown in Fig. 42, three faults originating from three different vehicles were detected.

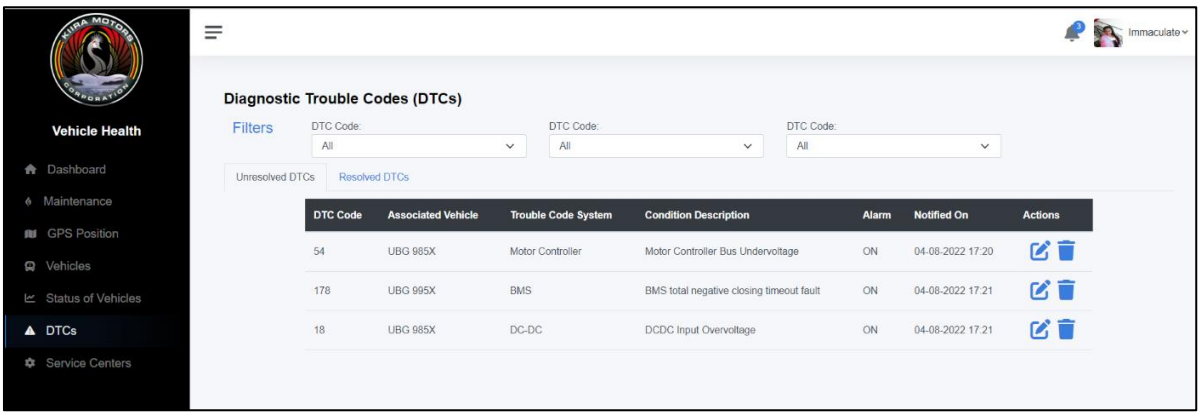


Figure 42: Detected faults in the vehicle

(vii) Service Centers

There are different service stations to avoid congestion and delay in services. Figure 43 indicates the station, the description of services to be done, and the coordinates for directions.

Most of the maintenance usually done are inspections and replacement types depending on how critical the fault is.

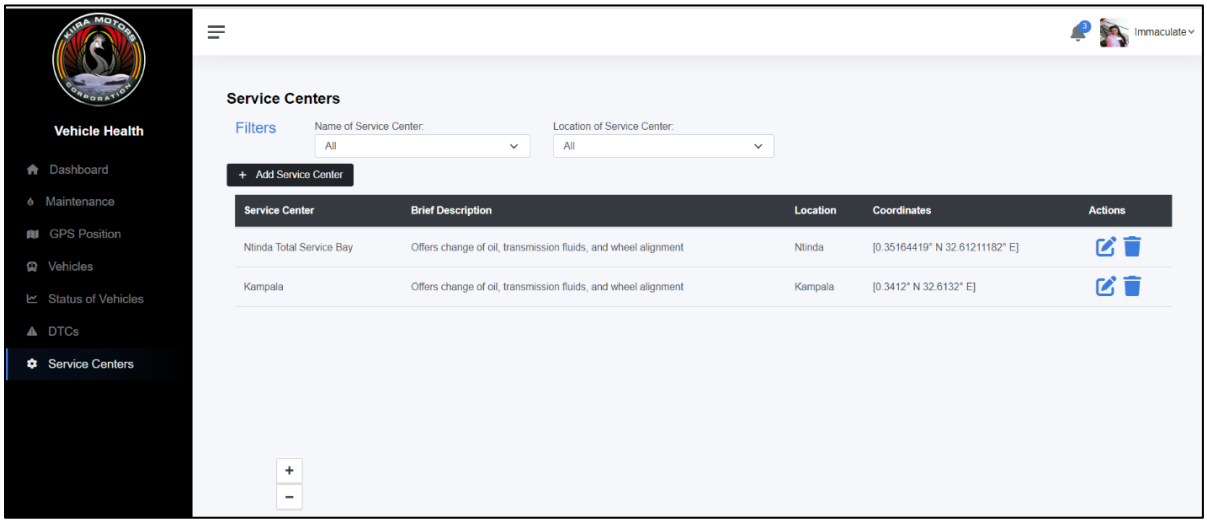


Figure 43: Available service centres

4.2 Discussion

The research project successfully demonstrated the effectiveness of an embedded device prototype and web application for remote monitoring of vehicle health status. The system acquired data from the vehicle's CAN communication network, transmitted the data wirelessly to the web application, and provided real-time monitoring of the vehicle's health status. The positive results achieved during the testing and evaluation of the prototype and web application are significant to attain the research questions and objectives of developing a reliable and efficient way to monitor the health of electric vehicles. However, the limitations and challenges encountered during the development and testing phases due to limited resources, time constraints, and the need for a stable internet connection highlight further research to optimize the system's performance and improve the user interface. Overall, the project has demonstrated the feasibility and effectiveness of the embedded device prototype and web application for remote monitoring of vehicle health status.

CHAPTER FIVE

CONCLUSION AND RECOMMENDATIONS

5.1 Conclusion

In conclusion, this research project aimed to develop a remote vehicle health monitoring system for the Kayoola electric vehicles integrated into Kiira EV. The project successfully achieved its objectives by developing an embedded system prototype and a web application for remote monitoring of vehicle health status. The system was designed to receive data from the vehicle's CAN communication network and transmit it wirelessly to the web application database. The developed system using Arduino Nano IoT 33 and MCP2515 CAN BUS module used to form the embedded device has the potential to increase vehicle safety, reduce maintenance costs, and improve overall vehicle performance. However, the technology needs further testing, validation, cybersecurity measures, more user-friendly interfaces from vehicle to vehicle, and ongoing support and maintenance to ensure its effectiveness and widespread adoption. Addressing these considerations can optimize the system's capabilities and improve its potential benefits for vehicle owners and the KMC as an automotive company. Overall, this project contributes to the growing field of IoT and smart transportation systems and offers a practical solution for remote monitoring of electric vehicles' health status.

5.2 Recommendations

(i) Stakeholders, such as manufacturers and fleet managers, should consider adopting the developed remote vehicle health monitoring system for their electric vehicles. The system's ability to remotely monitor key vehicle parameters can help prevent and diagnose issues, improving vehicle safety, reliability, and performance.

(ii) Further testing and validation should be conducted before implementing the system on a larger scale. The system was only tested and integrated into the Kiira EV. This will help identify and address potential issues and improve the system's effectiveness.

(iii) Advanced security measures, such as encryption, authentication and access controls, should be used by the system’s technical operations to protect the system from unauthorized access and ensure the privacy of the data collected. As a result, they are implementing advanced security measures.

(iv) To ensure the long-term effectiveness of the system, its technical operators should provide ongoing support and maintenance, including software updates, troubleshooting, and user training.

From the study the following recommendations for the future work are made:

(i) Further research should focus on developing machine learning algorithms to analyze the data collected by the system and provide insights for predictive maintenance.

(ii) The system can be expanded to include remote firmware updates for the vehicle's electronic control unit (ECU) to improve performance and address potential security vulnerabilities.

(iii) Integrating the system with smart transportation systems can enable real-time traffic monitoring and route optimization for electric vehicles. In addition, the system can monitor internal and external vehicle health faults and accident causes.

(iv) Also, the system can be adapted for use with other types of vehicles, such as hybrid and autonomous vehicles, to provide real-time health status monitoring.

Overall, the remote vehicle health monitoring system using Arduino Nano IoT 33 and MCP2515 CAN BUS module can be a valuable tool for improving vehicle safety and performance. Furthermore, the system can be optimized for widespread adoption and effectiveness by addressing the above recommendations.

REFERENCES

Africa-press. (2022). *Test Kampala taxis for roadworthiness*. https://www.africa-press.net/uganda/all-news/test-kampala-taxis-for-roadworthiness

AutoPi. (2022). *SAE J1939: The Ultimate Guide (2022)*. AutoPi. https://www.autopi.io/blog/j1939explained.

Bánhelyi, B., & Szabó, T. (2020). Data mining and analysis for data from vehicles based on the obdii standard. *Proceedings of the 11th International Conference on Applied Informatics*, 30-37.

CopperHillTechnologies. (2022). *A Brief Introduction to the SAE J1939 Protocol*. https://copperhilltech.com/a-brief-introduction-to-the-sae-j1939-protocol.

Digite (2022). *What Is Extreme Programming (XP)? & It's Values, Principles, And Practices*. https://www.digite.com/agile/extreme-programming-xp.

Ezhilarasu, C. M., Skaf, Z., & Jennions, I. K. (2019). The application of reasoning to aerospace Integrated Vehicle Health Management: Challenges and opportunities. *Progress in Aerospace Sciences*, *105*, 60-73.

Force, U. P. (2022). *Report of Accidents that Occurred from 02nd July to 09th July 2022*. https://www.upf.go.ug/press-release-report-of-accidents-that-occurred-from-2nd july-to-09th-july-2022.

Fritzing. (2022). *Fritzing.*

Ganesan, R., & Mydhile, S. (2013). Design and development of remote vehicle health monitoring system using context-aware web services. In *2013 IEEE Conference on Information & Communication Technologies*, 760-764.

Guru99. (2022). *Embedded Systems Tutorial: What is, History & Characteristics*. https://www.guru99.com/embedded-systems-tutorial.html.

Han, J., Kamber, M., & Pei, J. (2012). Data mining concepts and techniques third edition. *University of Illinois at Urbana-Champaign Micheline Kamber Jian Pei Simon Fraser University*. http://site.ebrary.com/id/10483440.

IoT, A. N. 33. (2022). *Arduino Nano 33 IoT datasheet* (2021st ed.). https://docs.arduino.cc/resources/datasheets/ABX00027-datasheet.pdf.

Kiiramotors (2022). *Who We Are*. kiiramotors. https://www.kiiramotors.com/about.

Lin, C. E., Li, C., Yang, S., & Lin, S. (2005). Development of On-Line Diagnostics and Real-Time Early Warning Systems for Vehicles. *2005 Sensors for Industry Conference*, 45–51. https://doi.org/10.1109/SICON.2005.257868.

Microchip. (2019). *MCP2515-Stand-Alone-CAN-Controller-with-SPI-20001801J*. https://ww1.microchip.com/downloads/en/DeviceDoc/MCP2515-Stand-Alone-CAN-Controller-with-SPI-20001801J.pdf.

Microchip. (2019-b). *Standalone CAN Controller with SPI Interface*. https://ww1.microchip.com/downloads/en/DeviceDoc/MCP2515-Stand-Alone-CAN-Controller-with-SPI-20001801J.pdf.

Microchip. (2019-c). *Standalone CAN Controller with SPI Interface*. www.microchip.com.

Momin, G. G., Purkar, A. P., Lokhande, N. S., Sayyad, A. A., & Chavhan, R. R. (2020). A Review on Vehicle Health Monitoring System. *International Journal for Modern Trends in Science and Technology*, *06*, 31–34.

Patel, J. (2021). *List of 10 Best Vuejs Frameworks to Use in 2022*. https://www.monocubed.com/blog/vuejs-frameworks.

Perişoară, L. A., Stamati, E. M., Chiţu, L. R., & Săcăleanu, D. L. (2018). Pilot Platform for Remote Monitoring of an Electric Vehicle. In *2018 IEEE 24th International Symposium for Design and Technology in Electronic Packaging,* 394-397.

Quintagroup (2022). *Django Rest Framework*. https://quintagroup.com/cms/python/django-rest-framework.

Shafi, U., Safi, A., Shahid, A. R., Ziauddin, S., & Saleem, M. Q. (2018). Vehicle remote health monitoring and prognostic maintenance system. *Journal of Advanced Transportation, 2018,* 1-10. https://doi.org/10.1155/2018/8061514.

Singh, S., Singh, A. K., & Sharma, A. (2021). OBD - II based Intelligent Vehicular Diagnostic System using IoT. *International Semantic Intelligence Conference, 21*, 511-515.

Softwaretestinghelp (2022). *Types of Software Testing: Different Testing Types with Details*. https://www.softwaretestinghelp.com/types-of-software-testing.

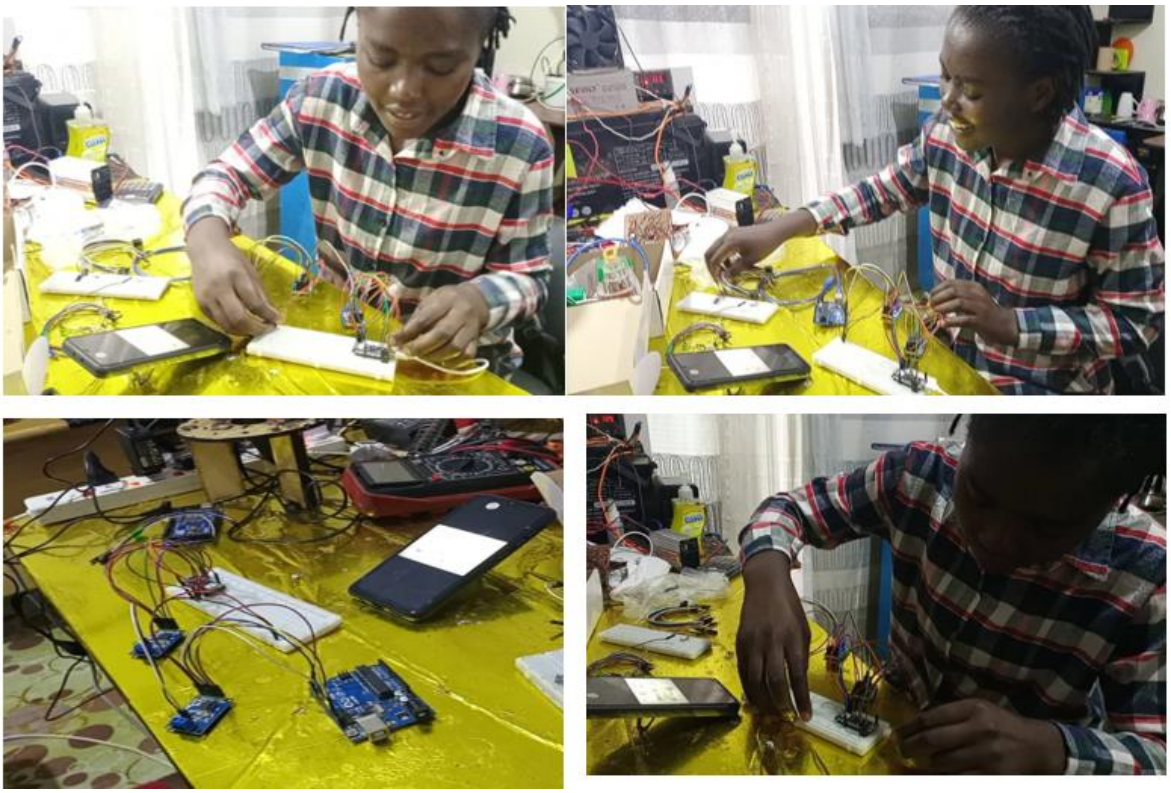
Technologies, S. (2022). *The Four Levels of Software Testing*. https://www.seguetech.com/the-four-levels-of-software-testing.

The Uganda Radio Network, U. R. N. (2022). *Uganda: 12 killed daily in road traffic accidents in past month*. The Independent. https://www.independent.co.ug/uganda-12-killed-daily-in-road-traffic-accidents-in-past-month.

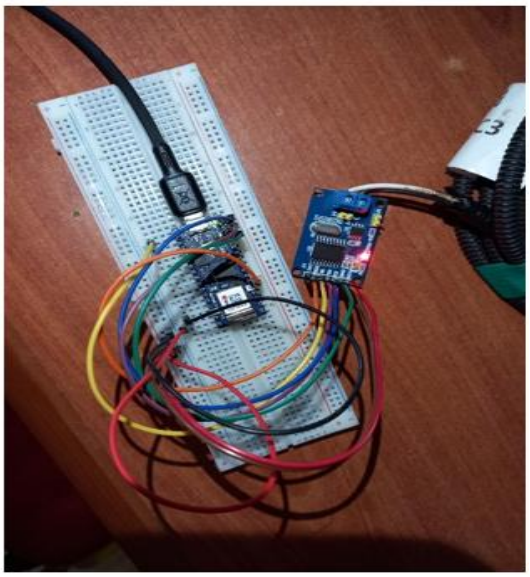
Trio (2022). *What Is Vue.js? The Pros and Cons of Vue.js*. https://www.trio.dev/blog/why-use-vue-js.

APPENDICES

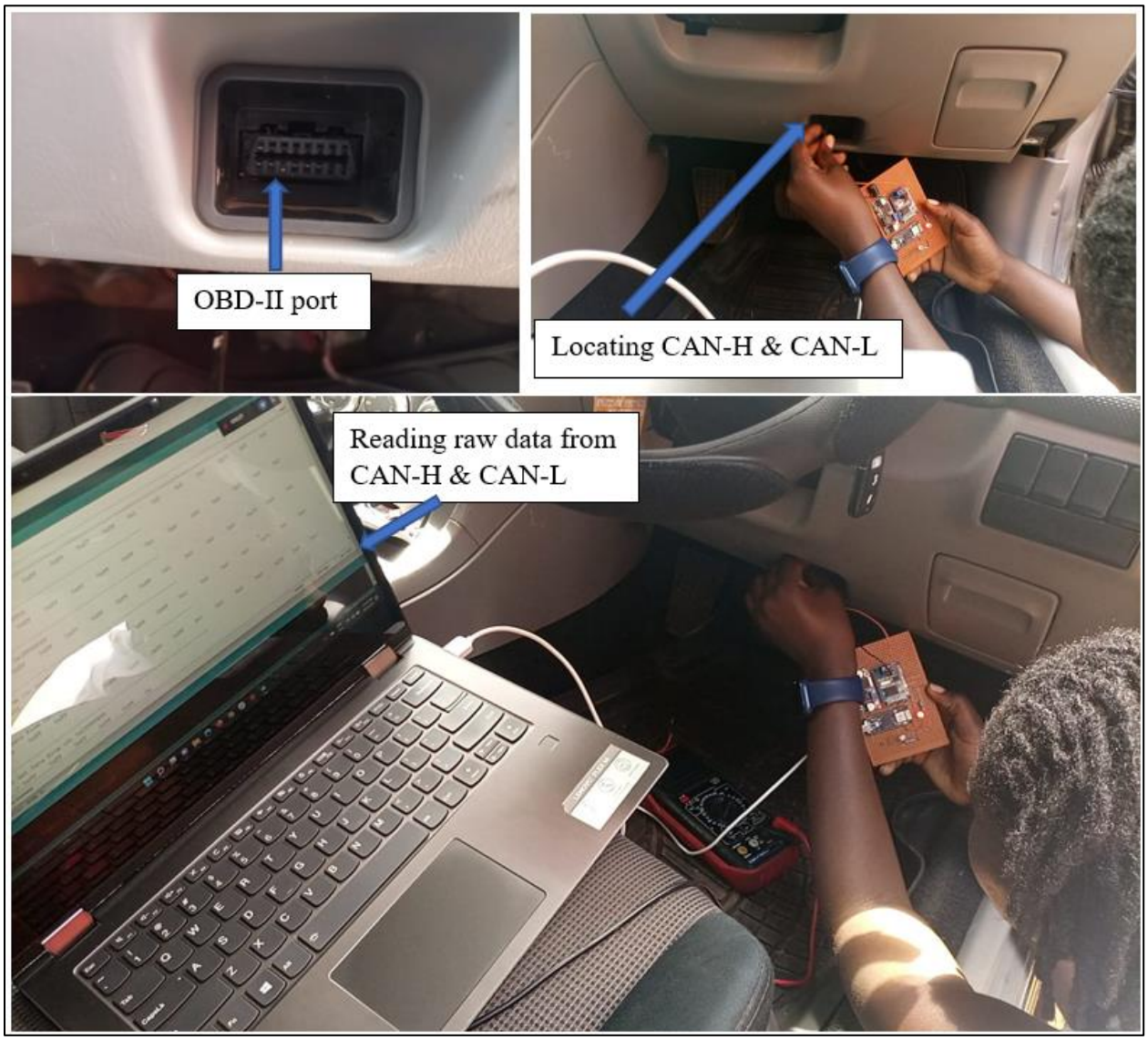
Appendix 1: Offboard Set up of the Hardware Set up



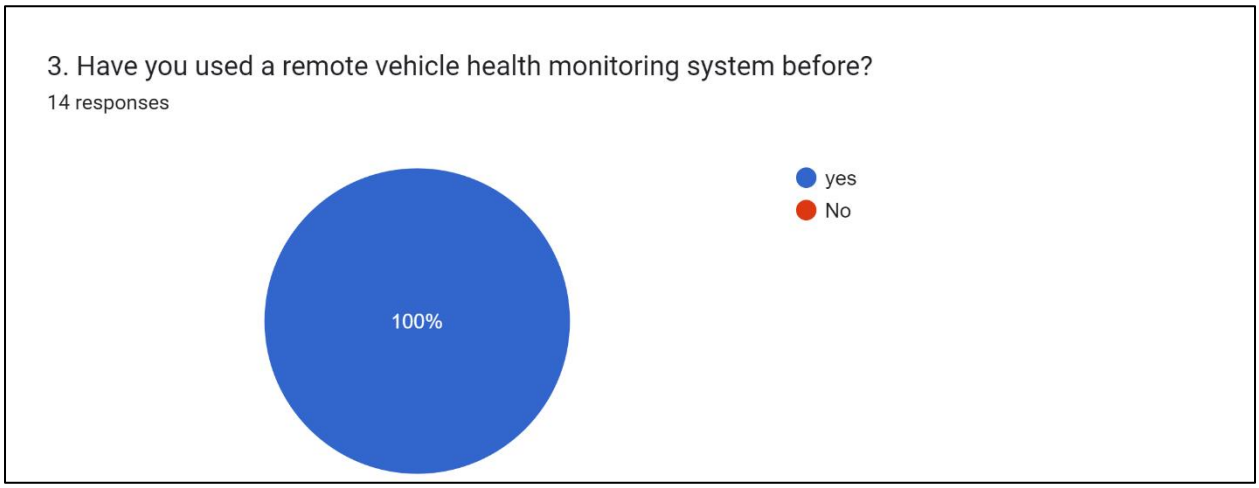
Appendix 2: Offboard Testing of the Hardware

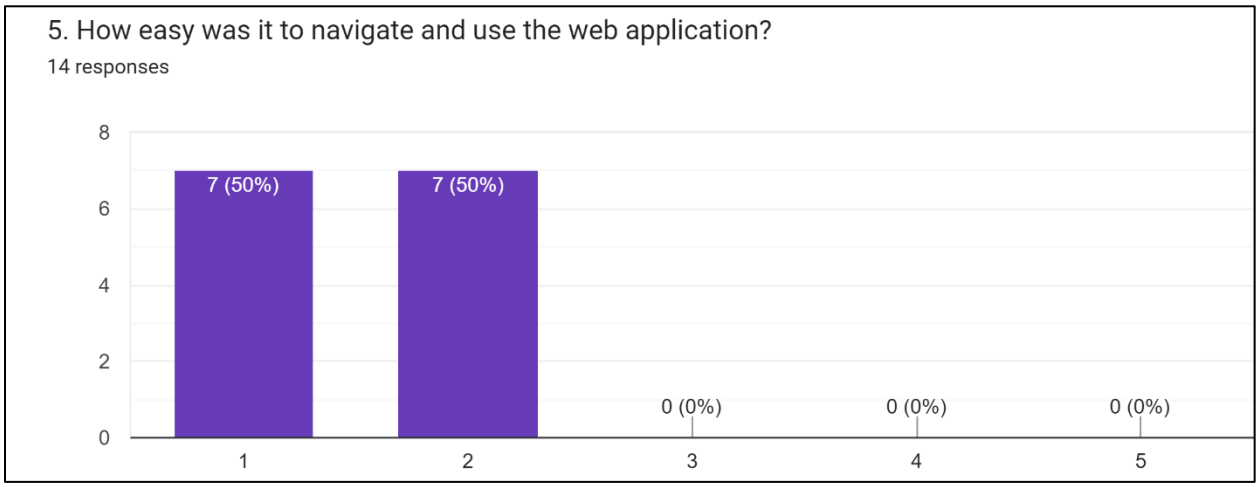


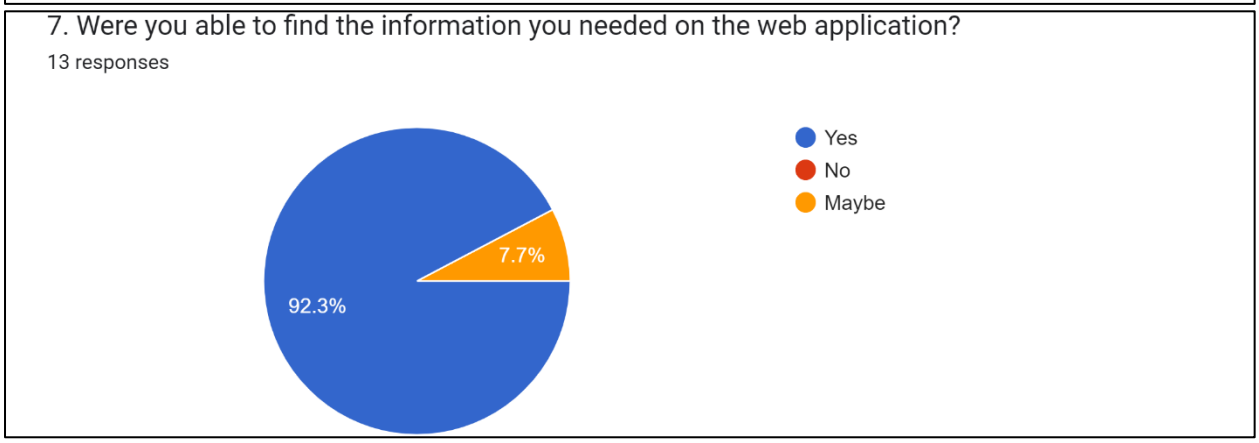
Appendix 3: Prototype Integration into the Kiira EV



Appendix 4: Validation Results







9. Were there any features or functionality that you would like to see added to the system?

5 responses

Driver's behaviors

Ability to detect when an accident is likely to happen

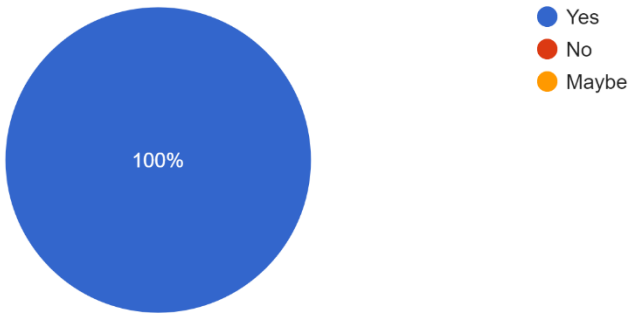
Enable the system to schedule maintenance

most yes

Accident detection

10. Would you recommend this remote vehicle health monitoring system to others?

14 responses



Appendix 5: Arduino Code for Acquiring Data from the Kiira EV

```
#include <ArduinoHttpClient.h>
#include <WiFi.h>
#include <HTTPClient.h>
#include <mcp_can.h>
#include <SPI.h>
#include "secrets.h"
//#include "ThingSpeak.h" // always include thingspeak header file after
other header files and custom macros

char ssid[] = SECRET_SSID;   // your network SSID (name)
char pass[] = SECRET_PASS;   // your network password
int keyIndex = 0;            // your network key Index number (needed only
for WEP)
WiFiClient  wifi;

//unsigned long myChannelNumber = SECRET_CH_ID;
//const char * myWriteAPIKey = SECRET_WRITE_APIKEY;

const char serverName[] = "kayoolaapi.pythonanywhere.com";  // server name
int port = 80;
HttpClient client = HttpClient( wifi, serverName, port );
int status = WL_IDLE_STATUS;

long unsigned int rxId;
unsigned char len = 0;
unsigned char rxBuf[8];
char msg387[128];
char msg393[128];                          // Array to store serial string
char msg165[128];
char msg389[128];                          // Array to store serial string
char msg166[128];
char msg167[128];                          // Array to store serial string
char msg171[128];
float initialTime = 0;

#define CAN0_INT 21                              // Set INT to pin 2
MCP_CAN CAN0(15);                               // Set CS to pin 10


void setup() {
  initialTime = millis();
  Serial.begin(115200);  //Initialize serial
  while (!Serial);
  WiFi.mode(WIFI_STA);
  //ThingSpeak.begin(client);  // Initialize ThingSpeak

  // Connect or reconnect to WiFi,This line creates an instance of the
WiFiClient class,
  //which allows the program to connect to a server using the WiFi network.
  if(WiFi.status() != WL_CONNECTED){
    Serial.print("Attempting to connect to SSID: ");
    Serial.println(SECRET_SSID);
    while(WiFi.status() != WL_CONNECTED){
      WiFi.begin(ssid, pass); // Connect to WPA/WPA2 network. Change this
line if using open or WEP network
      Serial.print(".");
      delay(5000);
    }
```

```
    Serial.println("\nConnected.");
  }

  // Initialize MCP2515 running at 8MHz with a baudrate of 500kb/s and the
masks and filters disabled.
  if(CAN0.begin(MCP_ANY, CAN_500KBPS, MCP_8MHZ) == CAN_OK)
    Serial.println("MCP2515 Initialized Successfully!");
  else
    Serial.println("Error Initializing MCP2515...");

  CAN0.setMode(MCP_NORMAL);                     // Set operation mode to
normal so the MCP2515 sends acks to received data.

  pinMode(CAN0_INT, INPUT);                            // Configuring pin
for /INT input
}

void loop()
{
  if(!digitalRead(CAN0_INT))                         // If CAN0_INT pin is
low, read receive buffer
  {
    CAN0.readMsgBuf(&rxId, &len, rxBuf);    // Read data: len = data length,
buf = data byte(s)

    Serial.print("The rxID is: ");
    Serial.println(rxId);

    switch (rxId) {
      case 393:
        sprintf(msg393, "Standard_ID:  0x%.3lX   DLC:  %1d   Data:  0x%.2X
0x%.2X 0x%.2X 0x%.2X 0x%.2X 0x%.2X 0x%.2X 0x%.2X", rxId, len, rxBuf[0],
rxBuf[1], rxBuf[2], rxBuf[3], rxBuf[4], rxBuf[5], rxBuf[6], rxBuf[7]);

        break;
      case 165:
        sprintf(msg165, "Standard_ID:  0x%.3lX   DLC: %1d   Data:  0x%.2X
0x%.2X 0x%.2X 0x%.2X 0x%.2X 0x%.2X 0x%.2X 0x%.2X", rxId, len, rxBuf[0],
rxBuf[1], rxBuf[2], rxBuf[3], rxBuf[4], rxBuf[5], rxBuf[6], rxBuf[7]);

        break;
      case 389:
        sprintf(msg389, "Standard_ID:  0x%.3lX   DLC: %1d   Data:  0x%.2X
0x%.2X 0x%.2X 0x%.2X 0x%.2X 0x%.2X 0x%.2X 0x%.2X", rxId, len, rxBuf[0],
rxBuf[1], rxBuf[2], rxBuf[3], rxBuf[4], rxBuf[5], rxBuf[6], rxBuf[7]);

        break;
      case 166:
        sprintf(msg166, "Standard_ID:  0x%.3lX   DLC: %1d   Data:  0x%.2X
0x%.2X 0x%.2X 0x%.2X 0x%.2X 0x%.2X 0x%.2X 0x%.2X", rxId, len, rxBuf[0],
rxBuf[1], rxBuf[2], rxBuf[3], rxBuf[4], rxBuf[5], rxBuf[6], rxBuf[7]);

        break;
      case 167:
        sprintf(msg167, "Standard_ID:  0x%.3lX   DLC: %1d   Data:  0x%.2X
0x%.2X 0x%.2X 0x%.2X 0x%.2X 0x%.2X 0x%.2X 0x%.2X", rxId, len, rxBuf[0],
rxBuf[1], rxBuf[2], rxBuf[3], rxBuf[4], rxBuf[5], rxBuf[6], rxBuf[7]);

        break;
      case 171:
```

```
        sprintf(msg171, "Standard_ID:  0x%.3lX   DLC: %1d   Data:  0x%.2X
0x%.2X 0x%.2X 0x%.2X 0x%.2X 0x%.2X 0x%.2X 0x%.2X", rxId, len, rxBuf[0],
rxBuf[1], rxBuf[2], rxBuf[3], rxBuf[4], rxBuf[5], rxBuf[6], rxBuf[7]);
        break;
      case 387:
        sprintf(msg387, "Standard_ID:  0x%.3lX   DLC: %1d   Data:  0x%.2X
0x%.2X 0x%.2X 0x%.2X 0x%.2X 0x%.2X 0x%.2X 0x%.2X", rxId, len, rxBuf[0],
rxBuf[1], rxBuf[2], rxBuf[3], rxBuf[4], rxBuf[5], rxBuf[6], rxBuf[7]);
        break;
      default:
        Serial.println("Undesired message!");
      }

  }

  if (millis() - initialTime > 5000) {
    String payload = String("{\"msg387\":\"" + (String)msg387 +
"\",\"msg393\":\"" + (String)msg393 + "\",\"msg165\":\"" + (String)msg165 +
"\",\"msg389\":\"" + (String)msg389 + "\",\"msg166\":\"" + msg166 +
     "\",\"msg167\":\"" + (String)msg167 + "\",\"msg171\":\"" +
(String)msg171 + "\",\"vehicle_id\":\"UBG 985X" + "\"}");

    VehicleHealthStatus(payload);
    }

  Serial.println( "Wait 100 milliseconds" );
  delay( 10 );
}

void VehicleHealthStatus(String payload) {
  String contentType = "application/json";

  Serial.println("Posting...");
  Serial.println(payload);

  client.post( "/health/raw-can-data/", contentType, payload );

  // read the status code and body of the response
  int statusCode = client.responseStatusCode();
  Serial.print( "Status code: " );
  Serial.println( statusCode );
  String response = client.responseBody();
  Serial.print( "Response: " );
  Serial.println( response );
  }
```

Appendix 6: Python Code for Decoding the Received Data

```
import serial
import cantools.database
import requests
import sys
import time

if __name__ == '__main__':
    trouble= {'vehicle_id': 'UBG 985X',
              'code': '111',
              'trouble_code_system': 'BMS',
              'condition_description': 'This is a test request',
              'alarm_on': 'ON',
              }
    ev = {'vehicle_id': 'UBG 985X',
              'battery_charge':'5',
              'battery_health':'Good',
              'battery_temperature':'10.0',
              'run_hours':'10.0',
              'distance_covered':'70.0',
              'speed':'30.0', # rpm (o.31) of the motor
              'state':'Active',
              'battery_voltage':'60.0',
              'battery_current':'6',
              'motor_speed':'100.0',
              'lat':'0.31703',
              'lng':'31.15553',
              }
    db = cantools.database.load_file('message_construct.dbc')
    with serial.Serial() as ser:
        ser.baudrate = 115200
        ser.port = 'COM4'
        ser.open()
        t1 = time.time()
        while True:
            data = ser.readline()
            try:
                data = data.decode()
                data = data.split('  ')
                frame_id = eval(data[0].split(' ')[-1].strip())
                dlc = eval(data[1].split(' ')[-1].strip())
                data = data[2].split('  ')[-1].split(':')[-1].strip()
                data = data.split(' ')
                data = [eval(i.strip()) for i in data]
                data = bytearray(data)

                msg = db.decode_message(frame_id, data)
                #print(f'Received: {msg}')
                if frame_id == 393:
                    ev['battery_charge'] = msg['D7_SOC']
                    ev['battery_health'] = 'Good' if msg['D7_SOC'] > 50
else 'Bad'#battery_health
                if frame_id == 165:
                    ev['motor_speed'] = msg['D2_Motor_Speed']
                if frame_id == 389:
                    ev['battery_temperature'] = msg['D4_BattCellTempAvg']
                if frame_id == 166:
                    ev['battery_current'] = msg['D4_DC_Bus_Current']
                if frame_id == 167:
```

```
                    ev['battery_voltage'] = msg['D1_DC_Bus_Voltage']
                 if frame_id == 393:
                    ev['run_hours'] = msg['D3_PackAmpHours']

                elif frame_id == 393:
                    pass
                    #print(f"SOC is: {msg['D7_SOC']} percent")
                elif frame_id == 393:
                    ev['12 Battery'] = msg['D2_12VoltSupply']
                elif frame_id == 387:
                    ev['battery_voltage'] =
msg['D1_BattVoltage']
                elif frame_id == 167:
                    ev['DC Voltage'] = msg['D1_DC_Bus_Voltage']
                elif frame_id == 171:
                    trouble['Fault_code'] = msg['M171_Fault_Codes']
                    for i in msg:
                        code = f'PM00{i}'
                        tcs = 'Powertrain'
                        alarm = 'ON' if msg[i] == 1 else 'OFF'
                        trouble['code'] = code
                        trouble['trouble_code_system'] = tcs
                        trouble['alarm_on'] = alarm
                        if msg[i] == 1:
                            response =
requests.post('https://kayoolaapi.pythonanywhere.com/health/trouble-
codes/', json=trouble)
                            print(response.status_code)
                            print(response.json())
                        print(f"{i} is: {msg[i]} binary\n")
            except Exception:
                pass
            finally:
                if time.time() - t1 > 5:
                    t1 = time.time()
                    #print('Now updating the backend')
                    #r =
requests.post('https://kayoolaapi.pythonanywhere.com/health/engine-
diagnostics/post', data={'key': 'value'})


                    response =
requests.post('https://kayoolaapi.pythonanywhere.com/health/ev-
diagnostics/',json=ev)
                    print(response.status_code)
                    print(response.json())
```

Appendix 7: Python Code for Analyzing the Received Raw Data

```
import requests

def post(url, data):
        return requests.post(url, json=data)


if __name__ == '__main__':
    data = {'vehicle_id': 'UBG 985X',
            'code': '100',
            'trouble_code_system': 'BMS',
            'condition_description': 'This is a test request',
            'alarm_on': 'ON',
            }
    url = 'https://kayoolaapi.pythonanywhere.com/health/trouble-codes/'
  #  print('Posting')

    #response = post(url, data)
    #print(response.status_code)
    #print(response.json())

    data = {'vehicle_id': 'UBG 985X',
            'battery_charge':'5',
            'battery_health':'Good',
            'battery_temperature':'10.0',
            'run_hours':'10.0',
            'distance_covered':'70.0',
            'speed':'30.0',
            'state':'Active',
            'battery_voltage':'60.0',
            'battery_current':'6',
            'motor_speed':'100.0',
            'lat':'0.31703',
            'lng':'31.15553',
            }
    url = 'https://kayoolaapi.pythonanywhere.com/health/ev-diagnostics/'

    response = post(url, data)
    print(response.status_code)
    print(response.json())
```

Appendix 8: Python Code Transforming and Posting Results to the Web App

```
# parsing data from the client
from rest_framework.parsers import JSONParser
# To bypass having a CSRF token
from django.views.decorators.csrf import csrf_exempt
# for sending response to the client
from django.http import HttpResponse, JsonResponse
# API definition for task
from . import serializers
from . import models
import cantools.database

# Create your views here.
@csrf_exempt
def getRawCANData(request):
    if request.method == 'POST':
        # parse the incoming information

        db =
cantools.database.load_file('/home/Kayoolaapi/django_rest_api/static/messag
e_construct.dbc')
        codes= []
        ev = {'vehicle_id': 'UBG 985X',
                'battery_charge':'12.2',
                'battery_health':'Good',
                'battery_temperature':'10.0',
                'run_hours':'10.0',
                'distance_covered':'70.0',
                'speed':'30.0', # rpm (o.31) of the motor
                'state':'Active',
                'battery_voltage':'237.6',
                'battery_current':'0.0',
                'motor_speed':'100.0',
                'lat':'0.31703',
                'lng':'31.15553',
                }

        data_received = JSONParser().parse(request)
       # ev['vehicle_id'] = data_received['vehicle_id']

        for key in data_received:
            if key != "vehicle_id":
                data = data_received[key]
                if data != "":
                    data = data.split('   ')
                    frame_id = eval(data[0].split(' ')[-1].strip())
                    data = data[2].split('  ')[-1].strip()
                    data = data.split(' ')
                    data = [int(i.strip(), 16) for i in data]
                    data = bytearray(data)
                    msg = db.decode_message(frame_id, data)


                    if frame_id == 393:
                        ev['battery_charge'] = msg['D7_SOC']
                        ev['battery_health'] = 'Good' if msg['D7_SOC'] > 50
else 'Bad'#battery_health
                        ev['run_hours'] = msg['D3_PackAmpHours']
                    if frame_id == 165:
```

```
                        ev['motor_speed'] = msg['D2_Motor_Speed']
                        print("motor_speed = ", ev['motor_speed'])
                    if frame_id == 389:
                        ev['battery_temperature'] =
msg['D4_BattCellTempAvg']
                    if frame_id == 166:
                        ev['battery_current'] = msg['D4_DC_Bus_Current']
                    if frame_id == 167:
                        ev['battery_voltage'] = msg['D1_DC_Bus_Voltage']
                    if frame_id == 387:
                        ev['battery_voltage'] = msg['D1_BattVoltage']
                    elif frame_id == 171:
                        for i in msg:
                            code = f'PM00{i}'
                            tcs = 'Powertrain'
                            if msg[i] == 1:
                                codes.append({'vehicle_id': 'UBG 985X',
                                    'code': code,
                                    'trouble_code_system': tcs,
                                    'condition_description': 'This is a
test request',
                                    'alarm_on': 'ON'})

        evSerializer = serializers.EvDiagnosticSerializer(data=ev)
        vehicle =
models.Vehicle.objects.get(pk=evSerializer.initial_data['vehicle_id'])
        # check if the sent information is okay
        evResponse = None
        if evSerializer.is_valid():
            # if okay, save it on the database
            evSerializer.save(vehicle=vehicle)
            # provide a Json Response with the data that was saved
            evResponse = evSerializer.data
        else:
            evResponse = evSerializer.errors

        for i in codes:
            dtcSerializer = serializers.TroubleCodeSerializer(data=i,
context={'request': request})
            vehicle =
models.Vehicle.objects.get(pk=dtcSerializer.initial_data['vehicle_id'])
            # check if the sent information is okay
            if dtcSerializer.is_valid():
                # if okay, save it on the database
                dtcSerializer.save(vehicle=vehicle)
                # provide a Json Response with the data that was saved
                evResponse.update(dtcSerializer.data)
            else:
                evResponse.update(dtcSerializer.errors)

    print('The user posted: ', data_received)
    return JsonResponse(evResponse, status=201)

@csrf_exempt
def diagnostics(request):
    """
    List all task snippets
    """
    if request.method == 'GET':
        # get all the tasks
        report_data = models.EvDiagnostic.objects.all()
```

```
        # serialize the task data
        serializer = serializers.EvDiagnosticSerializer(report_data,
many=True)
        # return a Json response
        return JsonResponse(serializer.data, safe=False)
    elif request.method == 'POST':
        # parse the incoming information
        data = JSONParser().parse(request)
        # instantiate with the serializer
        serializer = serializers.EvDiagnosticSerializer(data=data)
        # check if the sent information is okay
        if serializer.is_valid():
            # if okay, save it on the database
            serializer.save()
            # provide a Json Response with the data that was saved
            return JsonResponse(serializer.data, status=201)
            # provide a Json Response with the necessary error information
        return JsonResponse(serializer.errors, status=400)


@csrf_exempt
def diagnostic_detail(request, pk):
    try:
        diagnostic = models.EvDiagnostic.objects.get(pk=pk)
    except:
        # respond with a 404 error message
        return HttpResponse(status=404)
    if request.method == 'PUT':
        # parse the incoming information
        data = JSONParser().parse(request)
        # instantiate with the serializer
        serializer = serializers.EvDiagnosticSerializer(diagnostic,
data=data)
        # check whether the sent information is okay
        if serializer.is_valid():
            # if okay, save it on the database
            serializer.save()
            # provide a JSON response with the data that was submitted
            return JsonResponse(serializer.data, status=201)
        # provide a JSON response with the necessary error information
        return JsonResponse(serializer.errors, status=400)
    elif request.method == 'DELETE':
        # delete the task
        diagnostic.delete()
        # return a no content response.
        return HttpResponse(status=204)


@csrf_exempt
def trouble_codes(request):
    """
    List all task snippets
    """
    if request.method == 'GET':
        # get all the tasks
        report_data = models.TroubleCode.objects.all()
        # serialize the task data
        serializer = serializers.TroubleCodeSerializer(report_data,
many=True, context={'request': request})
        # return a Json response
        return JsonResponse(serializer.data, safe=False)
```

```
    elif request.method == 'POST':
        # parse the incoming information
        data = JSONParser().parse(request)
        # instantiate with the serializer
        serializer = serializers.TroubleCodeSerializer(data=data,
context={'request': request})
        vehicle =
models.Vehicle.objects.get(pk=serializer.initial_data['vehicle_id'])
        # check if the sent information is okay
        if serializer.is_valid():
            # if okay, save it on the database
            serializer.save(vehicle=vehicle)
            # provide a Json Response with the data that was saved
            return JsonResponse(serializer.data, status=201)
            # provide a Json Response with the necessary error information
        return JsonResponse(serializer.errors, status=400)


@csrf_exempt
def trouble_code_detail(request, pk):
    try:
        # obtain the task with the passed id.
        code = models.TroubleCode.objects.get(pk=pk)
    except:
        # respond with a 404 error message
        return HttpResponse(status=404)
    if request.method == 'PUT':
        # parse the incoming information
        data = JSONParser().parse(request)
        # instantiate with the serializer
        serializer = serializers.TroubleCodeSerializer(code, data=data,
context={'request': request})
        # check whether the sent information is okay
        if serializer.is_valid():
            # if okay, save it on the database
            serializer.save()
            # provide a JSON response with the data that was submitted
            return JsonResponse(serializer.data, status=201)
        # provide a JSON response with the necessary error information
        return JsonResponse(serializer.errors, status=400)
    elif request.method == 'DELETE':
        # delete the task
        code.delete()
        # return a no content response.
        return HttpResponse(status=204)


@csrf_exempt
def vehicles(request):
    """
    List all task snippets
    """
    if request.method == 'GET':
        # get all the tasks
        report_data = models.Vehicle.objects.all()
        # serialize the task data
        serializer = serializers.VehicleSerializer(report_data, many=True)
        # return a Json response
        return JsonResponse(serializer.data, safe=False)
    elif request.method == 'POST':
        # parse the incoming information
```

```
        data = JSONParser().parse(request)
        # instantiate with the serializer
        serializer = serializers.VehicleSerializer(data=data)
        # check if the sent information is okay
        if serializer.is_valid():
            # if okay, save it on the database
            serializer.save()
            # provide a Json Response with the data that was saved
            return JsonResponse(serializer.data, status=201)
            # provide a Json Response with the necessary error information
        return JsonResponse(serializer.errors, status=400)


@csrf_exempt
def vehicle_detail(request, pk):
    try:
        # obtain the task with the passed id.
        vehicle = models.Vehicle.objects.get(pk=pk)
    except:
        # respond with a 404 error message
        return HttpResponse(status=404)
    if request.method == 'PUT':
        # parse the incoming information
        data = JSONParser().parse(request)
        # instantiate with the serializer
        serializer = serializers.VehicleSerializer(vehicle, data=data)
        # check whether the sent information is okay
        if serializer.is_valid():
            # if okay, save it on the database
            serializer.save()
            # provide a JSON response with the data that was submitted
            return JsonResponse(serializer.data, status=201)
        # provide a JSON response with the necessary error information
        return JsonResponse(serializer.errors, status=400)
    elif request.method == 'DELETE':
        # delete the task
        vehicle.delete()
        # return a no content response.
        return HttpResponse(status=204)


@csrf_exempt
def maintenance_schedules(request):
    """
    List all task snippets
    """
    if request.method == 'GET':
        # get all the tasks
        report_data = models.MaintenanceSchedules.objects.all()
        # serialize the task data
        serializer = serializers.MaintenanceScheduleSerializer(report_data,
many=True, context={'request': request})
        # return a Json response
        return JsonResponse(serializer.data, safe=False)
    elif request.method == 'POST':
        # parse the incoming information
        data = JSONParser().parse(request)
        # instantiate with the serializer
        serializer = serializers.MaintenanceScheduleSerializer(data=data,
context={'request': request})
```

```
        vehicle =
models.Vehicle.objects.get(pk=serializer.initial_data['vehicle_id'])
        service_center =
models.ServiceCenters.objects.get(pk=serializer.initial_data['service_cente
r'])
        # check if the sent information is okay
        if serializer.is_valid():
            # if okay, save it on the database
            serializer.save(vehicle=vehicle, service_center=service_center)
            # provide a Json Response with the data that was saved
            return JsonResponse(serializer.data, status=201)
            # provide a Json Response with the necessary error information
        return JsonResponse(serializer.errors, status=400)


@csrf_exempt
def maintenance_schedule_detail(request, pk):
    try:
        # obtain the task with the passed id.
        maintenance = models.MaintenanceSchedules.objects.get(pk=pk)
    except:
        # respond with a 404 error message
        return HttpResponse(status=404)
    if request.method == 'PUT':
        # parse the incoming information
        data = JSONParser().parse(request)
        # instantiate with the serializer
        serializer = serializers.MaintenanceScheduleSerializer(maintenance,
data=data, context={'request': request})
        # check whether the sent information is okay
        if serializer.is_valid():
            # if okay, save it on the database
            serializer.save()
            # provide a JSON response with the data that was submitted
            return JsonResponse(serializer.data, status=201)
        # provide a JSON response with the necessary error information
        return JsonResponse(serializer.errors, status=400)
    elif request.method == 'DELETE':
        # delete the task
        maintenance.delete()
        # return a no content response.
        return HttpResponse(status=204)


@csrf_exempt
def service_centers(request):
    """
    List all task snippets
    """
    if request.method == 'GET':
        # get all the tasks
        report_data = models.ServiceCenters.objects.all()
        # serialize the task data
        serializer = serializers.ServiceCenterSerializer(report_data,
many=True)
        # return a Json response
        return JsonResponse(serializer.data, safe=False)
    elif request.method == 'POST':
        # parse the incoming information
        data = JSONParser().parse(request)
        # instantiate with the serializer
```

```
        serializer = serializers.ServiceCenterSerializer(data=data)
        # check if the sent information is okay
        if serializer.is_valid():
            # if okay, save it on the database
            serializer.save()
            # provide a Json Response with the data that was saved
            return JsonResponse(serializer.data, status=201)
            # provide a Json Response with the necessary error information
        return JsonResponse(serializer.errors, status=400)


@csrf_exempt
def service_center_detail(request, pk):
    try:
        # obtain the task with the passed id.
        service_center = models.ServiceCenters.objects.get(pk=pk)
    except:
        # respond with a 404 error message
        return HttpResponse(status=404)
    if request.method == 'PUT':
        # parse the incoming information
        data = JSONParser().parse(request)
        # instantiate with the serializer
        serializer = serializers.ServiceCenterSerializer(service_center,
data=data)
        # check whether the sent information is okay
        if serializer.is_valid():
            # if okay, save it on the database
            serializer.save()
            # provide a JSON response with the data that was submitted
            return JsonResponse(serializer.data, status=201)
        # provide a JSON response with the necessary error information
        return JsonResponse(serializer.errors, status=400)
    elif request.method == 'DELETE':
        # delete the task
        service_center.delete()
        # return a no content response.
        return HttpResponse(status=204)


@csrf_exempt
def ev_diagnostics(request):
    """
    List all task snippets
    """
    if request.method == 'GET':
        # get all the tasks
        report_data = models.EvDiagnostic.objects.all()
        # serialize the task data
        serializer = serializers.EvDiagnosticSerializer(report_data,
many=True, context={'request': request})
        # return a Json response
        return JsonResponse(serializer.data, safe=False)
    elif request.method == 'POST':
        # parse the incoming information
        data = JSONParser().parse(request)
        # instantiate with the serializer
        serializer = serializers.EvDiagnosticSerializer(data=data,
context={'request': request})
        vehicle =
models.Vehicle.objects.get(pk=serializer.initial_data['vehicle_id'])
```

```
        # check if the sent information is okay
        if serializer.is_valid():
            # if okay, save it on the database
            serializer.save(vehicle=vehicle)
            # provide a Json Response with the data that was saved
            return JsonResponse(serializer.data, status=201)
            # provide a Json Response with the necessary error information
        return JsonResponse(serializer.errors, status=400)


@csrf_exempt
def ev_diagnostic_detail(request, pk):
    try:
        # obtain the task with the passed id.
        diagnostic = models.EvDiagnostic.objects.get(pk=pk)
    except:
        # respond with a 404 error message
        return HttpResponse(status=404)
    if request.method == 'PUT':
        # parse the incoming information
        data = JSONParser().parse(request)
        # instantiate with the serializer
        serializer = serializers.EvDiagnosticSerializer(diagnostic,
data=data, context={'request': request})
        # check whether the sent information is okay
        if serializer.is_valid():
            # if okay, save it on the database
            serializer.save()
            # provide a JSON response with the data that was submitted
            return JsonResponse(serializer.data, status=201)
        # provide a JSON response with the necessary error information
        return JsonResponse(serializer.errors, status=400)
    elif request.method == 'DELETE':
        # delete the task
        diagnostic.delete()
        # return a no content response.
        return HttpResponse(status=204)


@csrf_exempt
def engine_diagnostics(request):
    """
    List all task snippets
    """
    if request.method == 'GET':
        # get all the tasks
        report_data = models.EngineDiagnostic.objects.all()
        # serialize the task data
        serializer = serializers.EngineDiagnosticSerializer(report_data,
many=True, context={'request': request})
        # return a Json response
        return JsonResponse(serializer.data, safe=False)
    elif request.method == 'POST':
        # parse the incoming information
        data = JSONParser().parse(request)
        # instantiate with the serializer
        serializer = serializers.EngineDiagnosticSerializer(data=data,
context={'request': request})
        vehicle =
models.Vehicle.objects.get(pk=serializer.initial_data['vehicle_id'])
        # check if the sent information is okay
```

```
        if serializer.is_valid():
            # if okay, save it on the database
            serializer.save(vehicle=vehicle)
            # provide a Json Response with the data that was saved
            return JsonResponse(serializer.data, status=201)
            # provide a Json Response with the necessary error information
        return JsonResponse(serializer.errors, status=400)


@csrf_exempt
def engine_diagnostic_detail(request, pk):
    try:
        # obtain the task with the passed id.
        diagnostic = models.EngineDiagnostic.objects.get(pk=pk)
    except:
        # respond with a 404 error message
        return HttpResponse(status=404)
    if request.method == 'PUT':
        # parse the incoming information
        data = JSONParser().parse(request)
        # instantiate with the serializer
        serializer = serializers.EngineDiagnosticSerializer(diagnostic,
data=data, context={'request': request})
        # check whether the sent information is okay
        if serializer.is_valid():
            # if okay, save it on the database
            serializer.save()
            # provide a JSON response with the data that was submitted
            return JsonResponse(serializer.data, status=201)
        # provide a JSON response with the necessary error information
        return JsonResponse(serializer.errors, status=400)
    elif request.method == 'DELETE':
        # delete the task
        diagnostic.delete()
        # return a no content response.
        return HttpResponse(status=204)
```

RESEARCH OUTPUTS

(i) Journal Paper

A Remote Vehicle Health Monitoring System for the Kayoola Electric Vehicle (Under review).

(ii) Poster Presentation

Appendix 9: Poster Presentation

A REMOTE EHICLE HEALTH MONITORING SYSTEM FOR THE KAYOOLA ELECTRIC VEHICLES

Immaculate Kamusiime[1], Emmanuel Masabo[1], Shubi Kaijage[1], Fred Matovu[2], Richard Madanda[2]
[1]The Nelson Mandela African Institution of Science and Technology, [2]Kiira Motors Corporation

Abstract

This paper presents a remote vehicle health monitoring system for Kayoola Electric Vehicles (EVs) manufactured by Kiira Motors Corporation (KMC) in Uganda. The system utilizes an embedded device and web application to monitor parameters such as voltage, current, temperature, speed, and location. Integration into the Kiira EV demonstrates the system's feasibility and effectiveness. The developed system enables remote tracking, monitoring, and notifications on the general health status of the EV based on live data. It aims to mitigate issues caused by the absence of proper vehicle health monitoring systems, such as resource waste, costly maintenance, and frequent breakdowns.

Introduction

Kiira Motors Corporation (KMC) in Uganda develops electric vehicles (EVs) and aims to establish a robust motor vehicle industry. However, the absence of a centralized vehicle health monitoring system leads to resource waste, expensive maintenance, and frequent breakdowns. This research addresses these concerns by developing a remote monitoring system for KMC's EVs. The system utilizes an embedded device and web application to track parameters and provide real-time data on the vehicles' health status.

Materials

The project utilized Arduino nano 33 IoT and MCP2515 CAN module for data acquisition from the vehicle's OBD-II port, with Arduino nano 33 IoT chosen for its small size and WiFi integration, and the MCP2515 CAN module selected for its CAN bus communication capabilities. Figure 1 is architecture

Results

They include developed embedded device prototype in figure 2 for wireless data transmission and a web app that decodes and displays real-time information on vehicle parameters and health status in figures 3 and 4.

Figure 3. Vehicle health Status

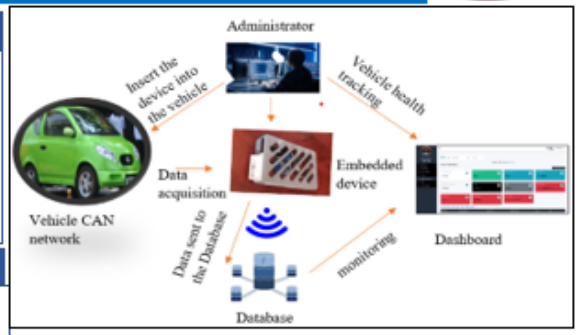


Figure 1. System Architecture

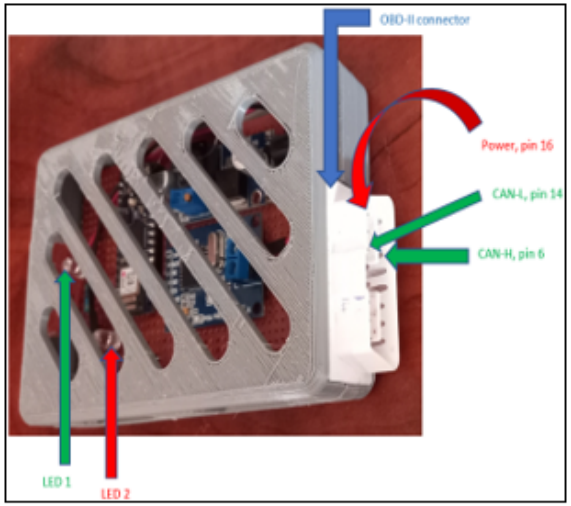


Figure 2. Embedded Device Prototype