Electrical Car's Digital Dashboard

By

Mohammed Rashad Nasr

Tasbeeh Khaled Mohammed

Refka Adel Ali

Saeed Magdy Shehata

Somia Hamdy Ali

Supervisor:

prof. Dr. Ayman Samy Abdel-Khalik

Undergraduate Project

Submitted to the Faculty of Engineering / Alexandria University as a partial fulfilment for BSc in [2022/2023]

Department of Electrical Power and Machines Engineering

Contents

List of Figures

Abstract

A car's digital dashboard is a system of Electronic Control Units (ECUs) that gathers information from various automotive components, such as the Battery Management System (BMS), motoring system, doors, etc. Then send the data to the Raspberry Pi, which is running a Linux OS-based QT application, using the CAN protocol. A QT application is created using Qtcreator and C++ as the back-end. C++ parses the data and prepares it for display before passing it to QML, which is the front-end language. Gauges, 3D objects, and notifications designed using QML are used to display and signal data changes in the QT application. The Yocto project is used to build and customize the Linux image to meet project requirements with the smallest possible footprint and greatest performance.

Acknowledgements

First and foremost, We want to acknowledge Prof. Dr. Ayman Samy for his outstanding mentorship and expertise. His profound knowledge, vast experience, and insightful guidance have played an instrumental role in shaping the success of our project. Prof. Dr. Ayman Samy's ability to provide clarity, offer strategic advice, and challenge us to think critically has been truly invaluable. His dedication to our growth, constant availability for discussions, and willingness to address our concerns have greatly enhanced our understanding of the subject matter and improved the quality of our work.

We would also like to extend my sincere gratitude to Eng. Ahmed Eldawy for his invaluable contributions and technical expertise. Eng. Ahmed Eldawy's deep understanding of the project's intricacies, attention to detail, and exceptional problem-solving skills have been pivotal in overcoming challenges and delivering high-quality results. His tireless efforts, availability to provide guidance, and commitment to excellence have truly made a difference in our project's outcomes.

Chapter 1

introduction

Due to constantly updated modern technologies that are very beneficial in many ways, digitalization is a widespread trend in many businesses and systems. Additionally, the automotive business is one of the sectors that receives upgrades and current technologies on a daily basis. The primary goal of applications in the automobile sector is to improve the driving experience by utilising tools like ADAS, HMI, safety systems, and navigation systems. All of these technologies are necessary to give the driver a sense of simple, dependable control over the numerous automotive systems in addition to safety and security considerations.

1.1 Vision and Goals

The use of microprocessors and other electrical devices will gradually make current cars smarter and more fuel-efficient. These electronic gadgets, known as Electrical Control Units (ECUs), are typically utilised to support the driver and make difficult decisions that are essential to safety and security. This project focuses on the design and development of a Car HMI incorporated into a cutting-edge completely electric vehicle. They are connected via CAN-BUS. The purpose of this project is to demonstrate a Car Dashboard that interactively shows real-time data about the vehicle, including speed, engine rpm, fuel, all car statuses (doors, tyres, etc.), GPS, IMUs, and battery-state messages received over the CAN bus. We will demonstrate how to construct a digital dashboard using a Raspberry Pi, a low-cost embedded platform,

and an optimised tool for graphical design. The finished product is inexpensive and easily adaptable to other vehicles in terms of both hardware and software. Future capabilities could be increased further. The project is aimed to be a fully customized product ready to compete in market with other well-known products. It also has the identity of Motovation team with the color scheme, logo and fonts so it's completely unique. We will also create our own Linux image to avoid using raspbian image which will not be suitable for a product. In addition, we are planning to find a any other board to fit project requirements without extra cost wasted on unused peripherals.

1.2 Similar Products

We've searched for cars in the market using digital dashboards for visual feed and to understand what kind of data should be displayed on the screen in addition to user experience concepts. Figures show examples of already released car dashboards.

Figure 1.1: Audi-A8

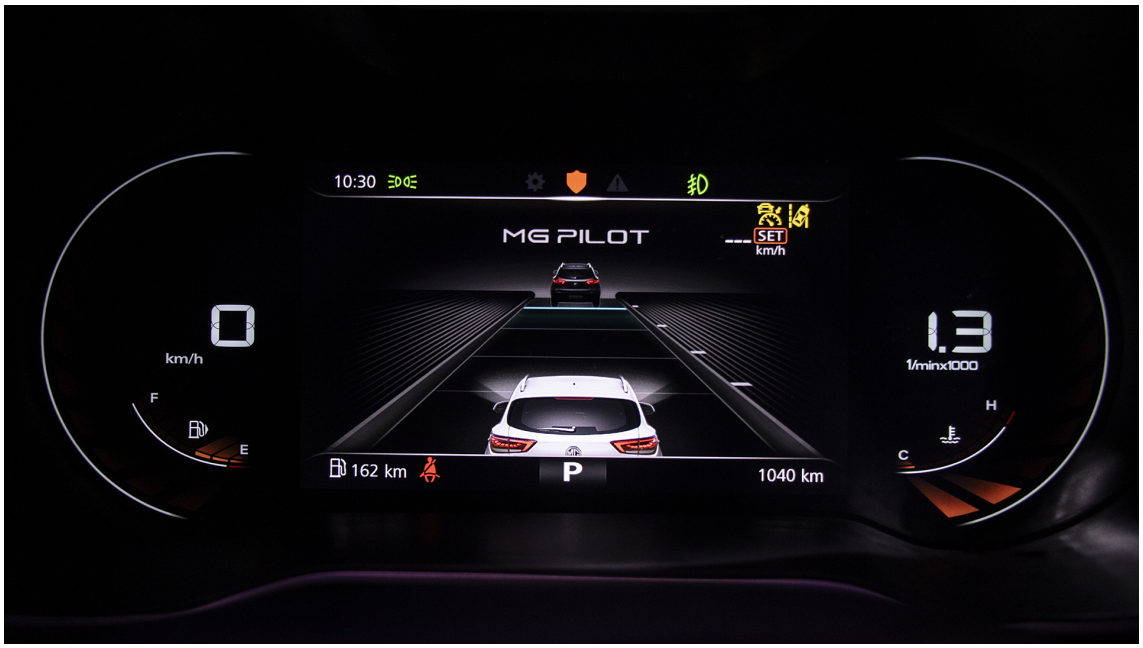

Figure 1.2: MG Astor

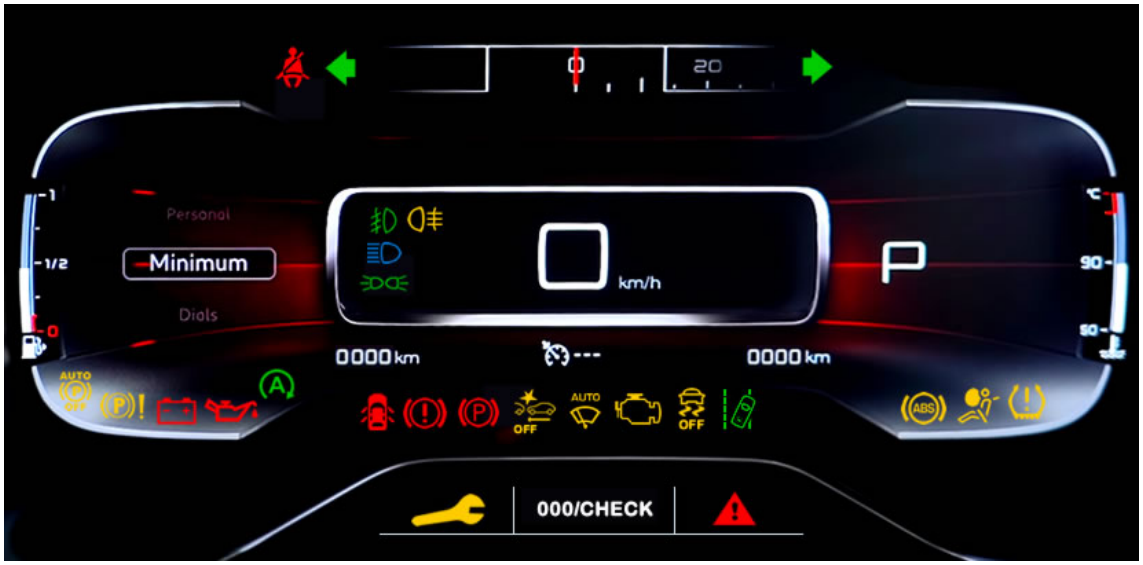

Figure 1.3: Citroen C5 Aircross

1.3 Methodolgy

The application will be developed and built using Qt platform which is a cross-platform usually used in such tasks. Qt is compatible with a multitude of operating systems and hardware. All you need is to write your source code once and have the capability to deploy it anywhere you need it. You do not need teams of developers coding specifically for different hardware architectures or operating systems.

Yocto project will be used to create our custom Linux image. The Yocto Project (YP) is an open source collaboration project that helps developers create custom Linux based systems regardless of the hardware architecture. The project provides a flexible set of tools and a space where embedded developers worldwide can share technologies, software stacks, configurations, and best practices that can be used to create tailored Linux images for embedded and IOT devices, or anywhere a customized Linux OS is needed.

Chapter 2

Back-end

2.1 CAN Protocol

The electric vehicle devices communicated with each other using CAN-bus, which is a critical component of this project. Therefore, it is essential to provide more detailed information about the CAN-bus. The subsequent sections will cover the programming of the CAN-bus.

2.1.1 Introduction

The Controller Area Network (CAN) protocol is a widely used communication protocol in the automotive industry for connecting different Electronic Control Units (ECUs) in a vehicle. It is a message-based protocol that allows for reliable and efficient data exchange between different devices. The CAN protocol operates on a two-wire bus, which allows for high-speed transmission of data between devices. The protocol is designed to handle multiple devices and supports prioritization of messages, ensuring that critical messages are delivered with high priority.

The integration of the CAN protocol in the dashboard system using a Raspberry Pi provides a powerful platform for collecting and processing data from different onboard systems. The Raspberry Pi is a single-board computer that offers high processing power and connectivity options, making it an ideal choice for implementing advanced dashboard systems in electric vehicles. The Raspberry Pi can communicate with different ECUs using the CAN protocol to collect real-time vehicle

information such as speed, battery status, and energy consumption.

One of the key advantages of the CAN protocol is its ability to handle multiple ECUs simultaneously. This means that the Raspberry Pi can communicate with different onboard systems in parallel, collecting and processing data from each system in real-time. This allows for the implementation of advanced features such as predictive maintenance, where the Raspberry Pi can collect and analyze data from multiple systems to predict when maintenance is required.

2.1.2 History

The Controller Area Network (CAN) protocol is a serial field bus protocol that was initially used in road vehicles. Its development history can be traced back to the early 1980s, when automotive manufacturers were using point-to-point wiring systems to connect different electronic control units (ECUs) in a vehicle. However, as the application of electronics increased rapidly, the wiring between different components became heavier, longer, and disorganized, making it expensive and difficult to repair. To address these challenges, Bosch developed the CAN protocol, which allows for reliable and efficient communication between different ECUs in a vehicle.

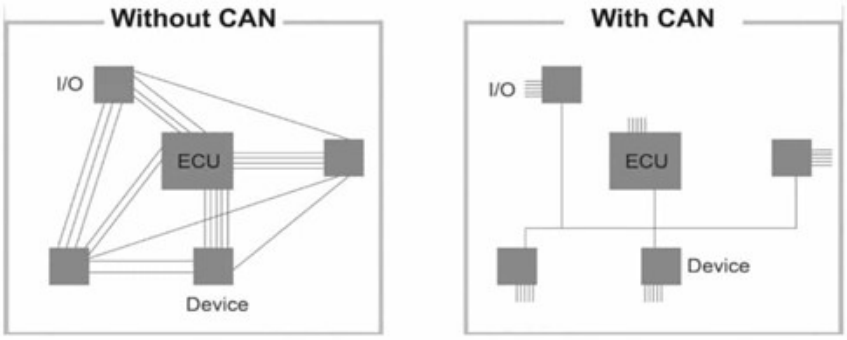

Figure 2.1: point-to-point vs CAN Communication

2.1.3 CAN bus layers

The CAN protocol is structured into different layers that work together to ensure reliable and efficient communication between different devices on the bus. The CAN protocol layers include the physical layer, data link layer, and application layer. The

physical layer is responsible for transmitting and receiving data on the CAN bus, while the data link layer is responsible for error detection and correction. The application layer is responsible for the exchange of data between different devices on the bus.

1. **Physical layer:**

   The physical layer in CAN communication is responsible for transmitting and receiving data on the CAN bus. The physical layer comprises two wires, namely the CAN High (CAN_H) and CAN Low (CAN_L) wires, that are twisted together to reduce electromagnetic interference. The CAN_H and CAN_L wires are used to transmit and receive differential signals, which are used to represent data on the bus. The physical layer also includes the transceiver, which converts the low-voltage digital signals from the CAN controller into high-voltage differential signals that can be transmitted on the bus. The transceiver also provides protection against electromagnetic interference and voltage spikes. The physical layer in CAN communication is critical for ensuring reliable and efficient communication between different devices on the bus.

2. **Data Link layer:**

   The data link layer in CAN communication is responsible for error detection and correction to ensure data integrity on the bus. The data link layer comprises two sub-layers, namely the logical link control (LLC) sub-layer and the media access control (MAC) sub-layer. The LLC sub-layer is responsible for flow control and framing, while the MAC sub-layer is responsible for arbitration and error detection. The CAN protocol uses a bit-wise arbitration scheme, where the highest priority message will win the arbitration and be transmitted on the bus. If two or more messages with the same priority are transmitted simultaneously, a collision will occur, and the messages will need to be re-transmitted. The data link layer in CAN communication also includes mechanisms for error detection and correction, such as Cyclic Redundancy Check (CRC) and Acknowledge (ACK) mechanisms. These mechanisms ensure data integrity and reliability in the transmission of data on the bus.

3. **Application layer:**

   The application layer in CAN communication is responsible for exchanging data between different devices on the bus. The application layer defines the format and structure of the messages exchanged between devices on the bus. The CAN protocol uses a message-based communication system, where each message consists of an identifier, data, and control information. The identifier is used to differentiate between different messages on the bus, while the data contains the actual information being transmitted. The control information includes details such as message length, priority, and error detection codes. The application layer in CAN communication also includes protocols for handling different types of messages, such as request-response messages, and broadcast messages.

2.1.4 CAN Frames

Frames are used to send messages via the CAN-bus, and these messages can be viewed by other devices on the bus. When a device sends a CAN frame, receiving devices will determine if they need to take any action based on the information contained in the frame. The CAN protocol supports four types of frames, including data frames, remote frames, error frames, and overload frames. Data frames are used to transmit data between different devices on the bus, while remote frames are used to request data from other devices. Error frames are used to indicate errors in transmission, while overload frames are used to indicate that a device is overloaded and cannot receive any more messages.

(a) **Data and Remote Frame**

The contents of a data frame include the following:

- **Start of frame (SOF)** This is a dominant bit that indicates the start of the frame. Arbitration field - This contains the identifier of the message and is used to determine which message has priority on the bus.

- **Control field** This contains information about the length of the message and any error detection codes that are being used.
- **Data field** This contains the actual data being transmitted between devices.
- **Cyclic Redundancy Check (CRC) field** This is used to detect errors in the transmission of the message.
- **Acknowledgment (ACK) field** This is used to acknowledge receipt of the message.
- **End of frame (EOF)** This is a recessive bit that indicates the end of the frame.

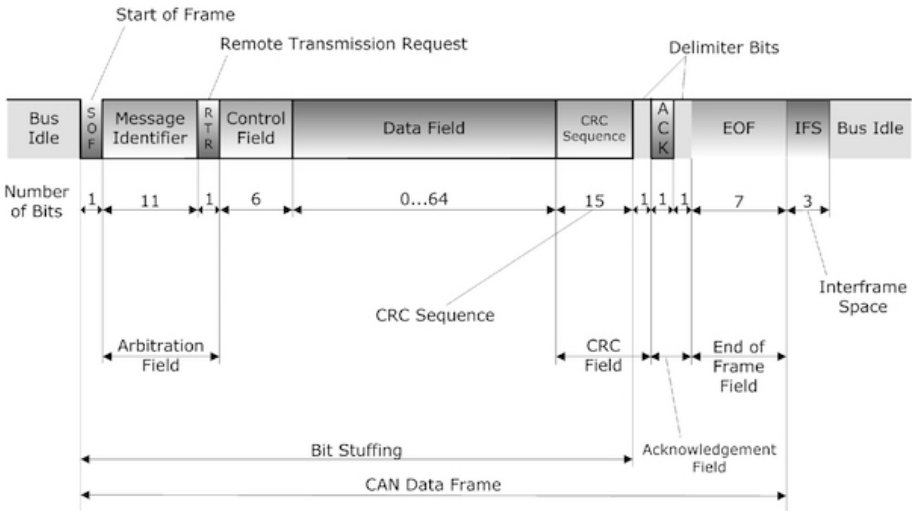

Figure 2.2: Standard 11-bit ID can Frame

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

(b) **Error Frame**

An error frame is a type of frame used in the Controller Area Network (CAN) protocol to indicate errors in transmission. When a device detects an error in a CAN frame, it will transmit an error frame to notify other devices on the bus that an error has occurred. The contents of the error frame include the error flags, Error State Indicator (ESI), and error delimiter. The error flags indicate the type of error that occurred, such as bit error, stuffing error, or frame error. The ESI bit indicates whether the

error was detected locally or remotely. The error delimiter is a recessive bit that indicates the end of the error frame.

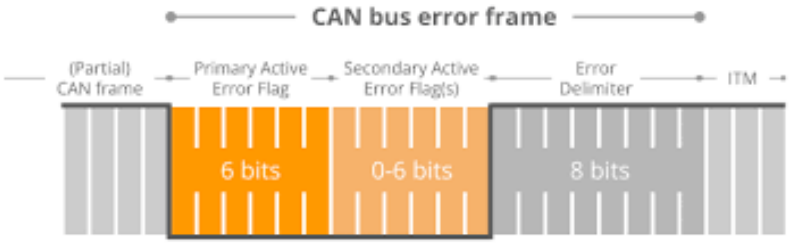

Figure 2.3: Error frame

(c) **Overload Frame**

An overload frame is a type of frame used in the Controller Area Network (CAN) protocol to indicate that a device is overloaded and cannot receive any more messages. When a device becomes overloaded, it will transmit an overload frame to notify other devices on the bus that it cannot receive any more messages. The contents of the overload frame include the overload flag, which is a recessive bit that indicates the start of the overload condition.

2.2 Components

Communication between various electronic devices is crucial in electric cars, and careful consideration was given to the selection of these devices. A significant criterion for their selection was their ability to be configured for inter-device communication. This chapter provides an overview of the chosen devices, and further details about them can be found in the provided resources.

2.2.1 Hardware components

- **Raspberry pi:**

  The core of the system is the Raspberry Pi 3 Model B running on the Raspbian buster operating system. This computer board is equipped with a 1.4GHz ARM Cortex-A53 processor and up to 1GB SRAM, in addition to integrated

Wi-Fi and Bluetooth functionalities. With 40 General Purpose Input/Output (GPIO) pins available, technical support and bug resolution, suitability for cross-compilation, relatively low cost, and support for a variety of hardware components such as HDMI monitors, touch screens, and cameras, it is more than enough for this project. Cross-compiling is the process of compiling software on one architecture, such as a desktop computer, for another architecture, such as Raspberry Pi's ARM architecture. Cross-compiling is useful because it allows us to build software on a more powerful machine and then deploy it to a less powerful device like Raspberry Pi.

- **STM32F103:**

  STM32F103 is a 32-bit ARM Cortex-M3 micro-controller that offers a wide range of features and capabilities. It is a popular choice for embedded systems and Internet of Things (IOT) applications due to its performance, efficiency, and low power consumption. The micro-controller has up to 128KB of Flash memory for code storage and up to 20KB of SRAM for data storage, making it suitable for projects that require a large amount of memory. Additionally, STM32F103 has a variety of built-in peripherals, including I2C, SPI, USART, CAN, and ADC, which allow easy integration with other hardware components. STM32F103 was chosen as the micro-controller for our project due to its advanced CAN options and capabilities. In our project, we needed a micro-controller that could efficiently handle CAN communication with other devices. The STM32F103 micro-controller has up to two CAN interfaces, which allowed us to easily integrate our project with other CAN-enabled devices. Additionally, the micro-controller's advanced CAN options, such as the ability to set up filters and masks for incoming messages, made it easier to manage and process CAN messages.

- **Arduino:**

  The Arduino Uno is a versatile micro-controller board that is suitable for a wide range of electronics projects, it is based on the Atmel ATmega328P micro-controller, which has 8-bit processing capabilities and 32KB of Flash memory for code storage. The micro-controller features a range of built-in peripherals,

including 14 digital input/output pins, 6 analog input pins, and 6 Pulse Width Modulation (PWM) output pins. Additionally, the Arduino Uno has a built-in USB interface for programming and communication with other devices. The micro-controller can be powered via USB or an external DC power supply, and it operates at 5V. We used it in our project because it's cost efficient and easy to use as another ECU in our car, for the CAN communication we used MCP2515 controller because Arduino doesn't have internal CAN peripheral.

- **Mcp2515 controller**

  The MCP2515 controller is a vital component in the system, serving as a stand-alone CAN controller that offers SPI communication with micro-controllers or other devices. It enables full-duplex communication with a maximum bit rate of 1 Mb/s, making it an excellent choice for high-speed data transfer. One of the key advantages of the MCP2515 is its configurability. It provides a wide range of configurable parameters, including baud rate, interrupt enable, and filter/mask configuration. This gives users greater flexibility in customizing the controller to suit the requirements of their specific application. Another significant feature of the MCP2515 is its support for the CAN 2.0B protocol. It provides error detection mechanisms such as CRC and ACK, ensuring reliable and secure communication between devices. In terms of technical details, the MCP2515 has a 20 MHz crystal oscillator and 8 MHz internal oscillator options. It has an operating voltage range of 2.7V to 5.5V and consumes very low power. The controller also has an 18-pin DIP/SOIC/MLF package and requires minimal external components for operation. There were many industrial shields we could use but this one was the only available one in Egypt, we did some modifications to it to use it with raspberry pi, it comes with CAN transceiver TJA1050 that works with 5 volts only which will burn the raspberry pi pins, so we removed and replaced it with SN65HVD230 which works with 3.3v.

- **Mcp2551 transceiver**

  MCP2551 is a high-speed CAN transceiver that was chosen for our project due to its advanced features and capabilities. Our project required a reliable

and efficient communication system that could handle data transmission over the CAN bus. The MCP2551 provided us with a highly reliable and robust solution that supported the ISO11898-2 standard, making it easy to interface with other CAN-enabled devices. The device's high-speed signaling capability allowed us to transmit data at rates up to 1 Mbps, ensuring fast and efficient data transmission over the bus. The MCP2551's low power consumption and low electromagnetic emissions (EMI) made it an ideal choice for our project, which required a reliable and efficient communication system that was also low on power consumption. Its robustness and EMI immunity ensured that our project was able to operate in a noisy environment without any interference.

2.2.2 SW components

- **Qt:**

  The user interface was created using a tool called Qt. Qt is a cross-platform framework, which means that applications programmed with it can run on different devices regardless of operating systems. It is used by high-profile companies such as Mercedes-Benz and Koenigsegg. Our project required a user interface that could communicate with other devices over the CAN bus, and Qt provided us with a comprehensive solution that allowed us to develop a sophisticated system quickly and efficiently. The back-end of our application was developed using C++, which provided us with a powerful and flexible programming language for handling the CAN communication and processing of data. Qt's CAN libraries enabled us to easily interface with the CAN bus, and the signals and slots mechanism allowed us to efficiently handle the incoming data from the bus and update the user interface accordingly. Additionally, the use of QML for the front-end allowed us to develop a modern and intuitive user interface that was easy to navigate and interact with. Qt was downloaded from the Qt official web-page https://www.qt.io/download. and selecting the open-source version. This method downloaded Qt online installer. The installer was used to select and download the right version of the Qt which was 5.15.0 in this case. The installer also installed Qt creator IDE which was used to

program the user interface.

- **Socket CAN:**

  SocketCAN is a powerful and flexible API for CAN communication in embedded systems and automotive applications. It provides a simple and efficient way to interface with CAN-enabled devices and develop applications for a wide range of industrial and automotive applications on the Raspberry Pi platform. The SocketCAN API provides a set of system calls for creating and configuring CAN interfaces, sending, and receiving CAN messages, and setting filters and masks to limit received messages based on specific criteria. Some of the key APIs are: socket(): Used to create a socket for CAN communication. bind(): Used to bind a socket to a CAN interface. Additionally, SocketCAN provides a range of utility programs for monitoring and analyzing CAN traffic, such as candump, which displays incoming CAN messages in real-time, and cansniffer, which provides a more detailed view of CAN traffic. It also provides utilities for sending and receiving CAN messages, such as cansend and cansequence. These tools make it easy to debug and analyze CAN communication and ensure that their applications are working as expected.

- **Can-utils:**

  Can-utils is a Linux-specific software package for interfacing the SocketCan driver of the Raspberry Pi kernel. To enable CAN communication on the Raspberry Pi, some additional configuration steps are required after installing the can-utils package. Firstly, the /boot/config.txt file needs to be modified by adding two lines at the end of the file: dtoverlay=mcp2515-can0, oscillator=8000000, Interrupt=12 and dtoverlay=spi-bcm2835-overlay. These lines enable the MCP2515 CAN controller on the Raspberry Pi by loading an overlay, specifying the CAN interface name, oscillator frequency, and interrupt GPIO pin, as well as enabling SPI communication. These changes enable the MCP2515 integrated circuit in the CAN shield instructions to function correctly. The CAN-network was set up using the command “sudo ip link set can0 up type can bitrate 250000” in the Raspberry’s terminal. This command sets the CAN network up at 250Kb/s speed.

2.2.3 Methodology

To simulate the CAN bus in a real car, we utilized multiple Electronic Control Units (ECUs) to send and receive data. This was because in an actual car, there are multiple ECUs responsible for different functions such as engine control, transmission control, and so on. By using multiple ECUs, we could simulate the behavior of a real car more accurately. For the micro-controller selection, we chose the STM32f103 because it is a popular micro-controller in the industry and has a built-in CAN peripheral. However, we developed custom CAN peripheral drivers from scratch. To facilitate communication between the ECUs, we used MCP2551 CAN transceivers. These transceivers protect the micro-controller from voltage spikes and other disturbances that can occur on the CAN bus. For the Raspberry Pi, which does not have an internal CAN peripheral, we used an MCP2515 controller. The MCP2515 is a CAN controller that interfaces with the Raspberry Pi via SPI. We chose this controller because it is widely available and has open-source drivers. For the software, we utilized the SocketCAN API in Qt. SocketCAN is a Linux-based CAN interface that provides a standardized API for accessing CAN devices. It allowed us to easily interface with the CAN controllers and transmit and receive CAN messages. We also incorporated an Arduino with an MCP2515 controller and utilized its library to facilitate communication. This allowed us to easily add more ECUs to the simulation if needed. Overall, our simulation system enabled us to replicate the functionality of the CAN bus in a real car, allowing us to test and verify our system's performance under various conditions. By utilizing a combination of hardware and software components, we were able to create a flexible and scalable simulation platform.

- **Hardware connection:**

figure 2.4 below shows that Our can bus in connected to 3 ECUS:

1. Raspberrypi connected using mcp2515 controller.
2. STM32F103 connected using mcp2551 transceiver and integrated with BMS.
3. Arduino connected using mcp2515 controller.

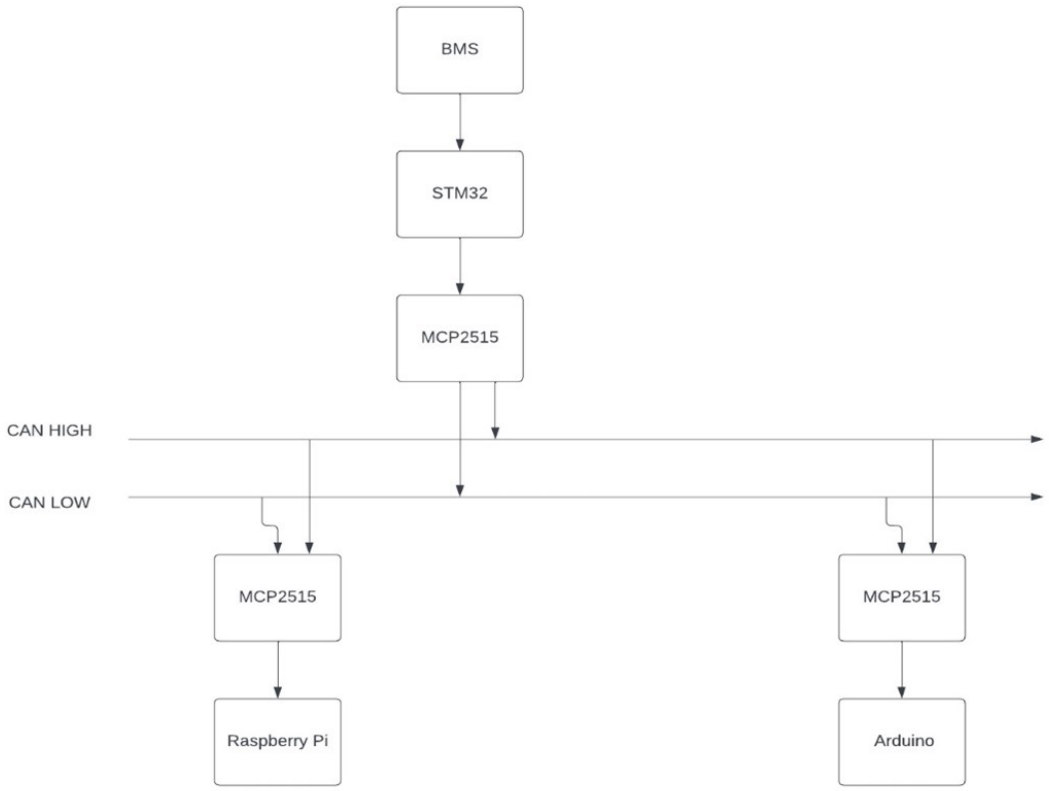

Figure 2.4: CAN bus hardware connection

- **software:**

1. **Qt libraries:**

   The communication between the user interface and the devices on the CAN bus was programmed using C++. As mentioned earlier Qt has many libraries and APIs to interface with the lower-level components and drivers of the target device.

   – **Qt Serial Bus**

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

   – **QCanBus**

     QCanBus class was used to connect the application to the socketCan driver of the raspberry. For example, the following call would connect to the SocketCAN interface vcan0:

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

- **QCanBusDevice**

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

- **QCanBusFrame**

  QCanBusFrame is a container class representing a single CAN frame. QCanBusDevice can use QCanBusFrame for read and write operations. It contains the frame identifier and the data payload.

2. **Signals and slots**

   Signals and slots are used for communication between objects. The signals and slots mechanism are a central feature of Qt and probably the part that differs most from the features provided by other frameworks. Signals and slots are made possible by Qt's meta-object system. We have an alternative to the callback technique: We use signals and slots. A signal is emitted when a particular event occurs. Qt's widgets have many predefined signals, but we can always subclass widgets to add our own signals to them. A slot is a function that is called in response to a particular signal. Qt's widgets have many pre-defined slots, but it is common practice to subclass widgets and add your own slots so that you can handle the signals that you are interested in. In our project, we utilized the signals and slots mechanism to connect the frameReceived signal to the checkFrame slot. As follows:

   ```
   QObject::connect(device, &QCanBusDevice::framesReceived, checkFrames);
   ```

   The frameReceived signal is emitted whenever a new message frame is received by our system. The checkFrame slot is responsible for reading the frame and

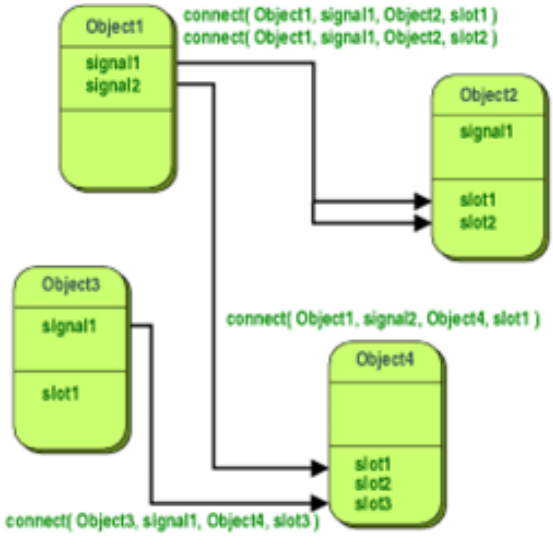

Figure 2.5: Signals and Slots concept

performing any necessary checks or processing. By utilizing signals and slots, we were able to decouple the logic for receiving and processing frames. This allowed us to easily modify or extend the system's behavior without affecting the other components. Additionally, signals and slots allowed us to establish a clear and intuitive communication flow between the components of our system.

3. **Integration between front-end and back-end**

   There are many methods for integrating between C++ back-end and QML front-end such as:

   (a) **Signals and Slots:** Signals and slots are a mechanism for communication between objects in Qt. Signals are emitted by an object when a particular event occurs, and slots are functions that are called in response to a signal. This mechanism is commonly used to integrate the front-end and back-end in Qt applications.

   (b) **Property Binding:** Property binding is a mechanism that allows the values of properties in one object to be automatically updated when the value of a connected property changes. This mechanism is useful for keeping the front-end and back-end in sync without the need for explicit communication.

   (c) **QML and C++ Integration:** Qt provides a declarative language called

QML for creating user interfaces. QML can be integrated with C++ back-end logic using the QObject class and the qmlRegisterType function. This integration allows for efficient communication between the front-end and back-end.

(d) **Direct Function Calls:** Direct function calls can be used to call C++ functions directly from QML. This mechanism is useful for simple operations that do not require complex communication between the front-end and back-end.

In our project, we used setProperty to integrate the C++ back-end and QML front-end. setProperty is a function provided by the QObject class in Qt that allows for the dynamic modification of object properties. This function can be used to set properties on QML objects from C++, which enables the integration of the two components. To use setProperty, we first defined the QObject classes in our C++ back-end and exposed them to QML Once the QML objects were created, we used setProperty to set their properties from the C++ back-end. For example, if we had a QML object with a property named batterylevel, we could set its value from the C++ back-end using the following code:

```
QObject *object;
object->setProperty("batterylevel", decodeFrame(frame,0));
```

4. **STM32F103 code**

The STM32F103 micro-controller features a built-in controller area network (CAN) module known as BXCAN. This module enables the micro-controller to communicate with other devices using the CAN protocol, The BXCAN module in STM32F103 supports both CAN 2.0A and CAN 2.0B protocols and is compatible with ISO 11898-1 and ISO 11898-2 standards. It features two CAN interfaces, each with a separate set of registers and filters. The module includes three transmit mailboxes and two receive FIFOs with three stages and scalable filter banks, which can be configured for different message types and priorities. The BXCAN module provides advanced error detection and correction mechanisms, such as bit monitoring, frame error detection, and

CRC checking. It also supports automatic retransmission of failed messages, as well as error status flags to notify the application of any errors. The message filtering system in the BXCAN module allows the micro-controller to accept or reject incoming messages based on their identifier, data content, and message type. The module supports both hardware and software filtering, which can be configured to match specific message patterns or ranges.

– **BXCAN Operating Modes:**

BXCAN supports three operating modes: sleep, initialization, and normal. In sleep mode, the BXCAN module is powered down and does not consume any current. The module transitions to sleep mode when the SLEEP bit in the CAN_MCR (Master Control Register) is set, to exit sleep mode the SLEEP bit is cleared by software or by hardware when the AWUM (Automatic Wakeup Mode) bit is set and the SOF bit is detected on the CAN Rx signal. In initialization mode, the module is configured with various settings and parameters before it enters normal mode. The initialization process includes configuring the bit timing, setting up the message filters, and enabling the interrupts. The initialization mode is entered by setting the INRQ (Initialization Request) bit in the CAN_MCR register. Once the initialization is complete, the module clears the INRQ bit and enters normal mode. In normal mode, the BXCAN module can operate in several different modes. To configure the mode of operation, the following registers are used: CAN_BTR (Bit Timing Register): This register sets the bit timing parameters, such as the propagation segment, phase segment 1, phase segment 2. CAN_FM1R (Filter Mode Register 1) and CAN_FM2R (Filter Mode Register 2): These registers are used to configure the message filters. The BXCAN module supports both hardware and software message filtering, and up to 28 filters can be configured. CAN_FMR (Filter Master Register): This register is used to enable or disable the message filters and set the filter scale (single or dual). CAN_IER (Interrupt Enable Register): This register is used to enable or disable the various interrupts generated by the BXCAN module, such as the receive FIFO interrupts,

transmit mailbox empty interrupts, and error interrupts.

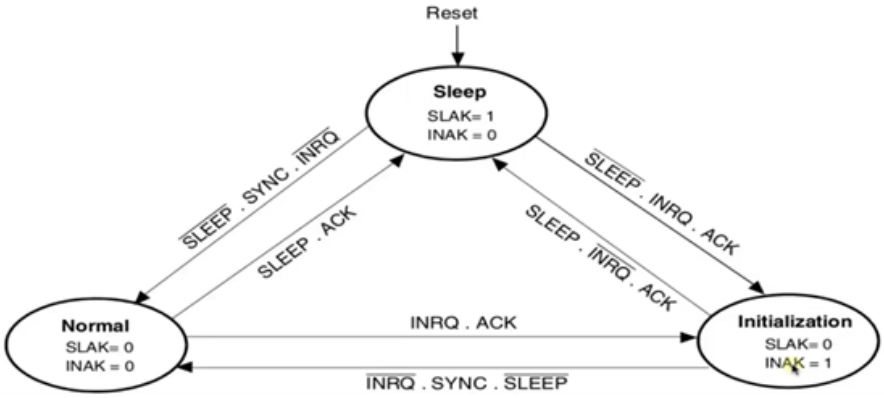

Figure 2.6: operating modes

– **Transmission Handling :**

The transmission system includes three transmit mailboxes that can be configured for different message types and priorities. To transmit a message, the application first writes the message data and identifier to one of the transmit mailboxes and sets the corresponding Transmit Request (TXRQ) bit in the CAN_TIxR register, where x is the mailbox number. Mailbox enters pending state and waits to become the highest priority mailbox, then it will be scheduled for transmission. Hardware indicates a successful transmission by setting RQCP and TXOK bits in the CAN_TSR register. If the transmission fails, the cause is indicated by the ALST bit in the CAN_TSR register in case of arbitration lost, or the TERR bit, in case of transmission error detection. As show in the figure below , the transmission process begins with an available empty mailbox. After writing the message data, identifier, and data length code (DLC), the transmission request is indicated by setting the TXRQ bit to 1, and the mailbox enters a pending state. If the mailbox has the highest priority, it enters a scheduled state. When the CAN bus is idle, it enters the transmission state. If the transmission is successful, the mailbox is emptied, and the RQCP and TXOK bits are set to 1. However, if the mailbox is in a pending or scheduled state, the transmission can be aborted

– **Bit Timing:**

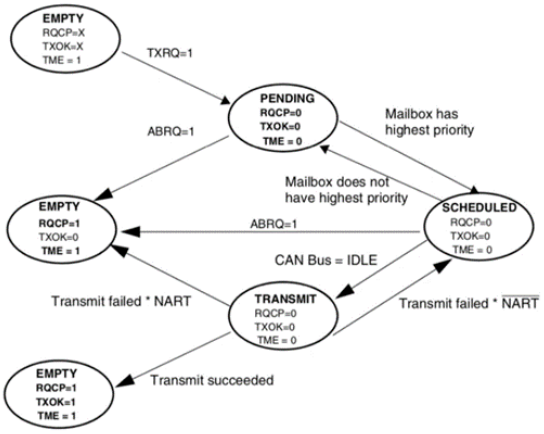

Figure 2.7: Transmission Handling

The bit rate on the CAN bus is determined by the bit time and the number of time quanta in a bit time. The bit rate can be calculated using the following formula: bit rate = CAN controller clock frequency / (prescaler * (1 + TS1 + TS2))

The synchronization segment is the first part of a bit time and is used to synchronize the CAN controller with the CAN bus. Its length is determined by the TS1 field. The propagation segment is the second part of a bit time and is used to compensate for signal propagation delays on the CAN bus. Its length is also determined by the TS1 field. The phase segments are the final parts of a bit time and are used to sample the bits of the message on the CAN bus. The length of the phase segments is determined by the TS2 field. The total number of time quanta in a bit time is determined by the sum of the TS1 and TS2 fields, plus one for the synchronization segment. The maximum bit rate on the CAN bus is typically limited to 1 Mbps, although slower bit rates can be used for longer bus lengths.

– **Reception Handling- FIFO management :**

In BXCAN, reception handling is managed using First-In-First-Out (FIFO) management techniques. When a message is received, it is placed in the receive FIFO buffer, which can store up to 3 messages at a time. The receive FIFO is managed by several registers, including the CAN_RFR register, which controls the release of the next message from the receive FIFO, and

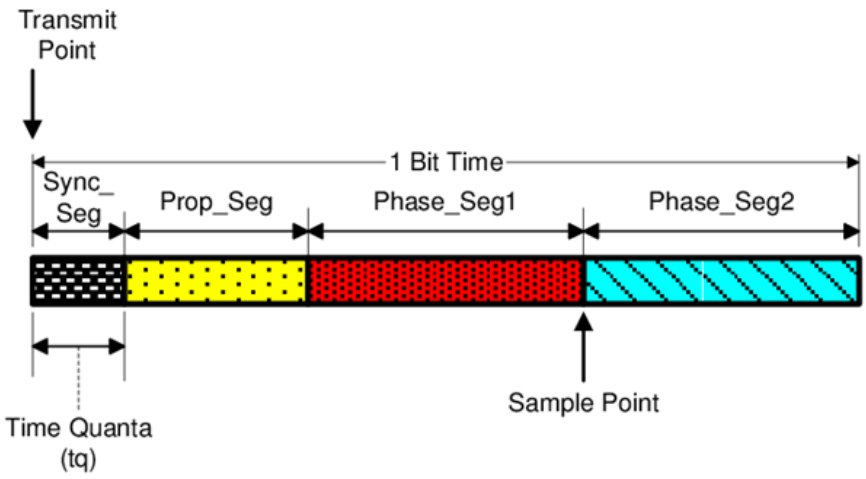

Figure 2.8: Bit Timing

the CAN_RIR and CAN_RDTR registers, which contain the identifier and data length of the received message, respectively. When a new message is received, it is placed at the end of the FIFO buffer, and the next message to be processed is always the oldest message in the buffer. The CAN_RFR register is used to release the next message from the receive FIFO, which then updates the CAN_RIR and CAN_RDTR registers with the identifier and data length of the released message. The released message can then be processed by the CAN controller. If the receive FIFO is full, and a new message is received, the oldest message in the FIFO buffer is overwritten with the new message, and the CAN_RFR register is updated accordingly. This ensures that the receive FIFO always contains the most recent messages.

– **Identifier filtering :**

BXCAN has 28 configurable filter banks (27-0), each filter bank consists of two 32 bit registers, CAN_FxR0 and CAN_FxR1. Each filter bank can be scaled independently: one 32 bit filter or two 16 bit filter. In BXCAN, message filtering is an important aspect of message reception, and there are two filtering modes available: mask mode and identifier list mode. In mask mode, filter banks are configured to accept messages that match a particular identifier or range of identifiers. The filter identifier specifies the bits of the identifier that must match exactly, while the filter mask

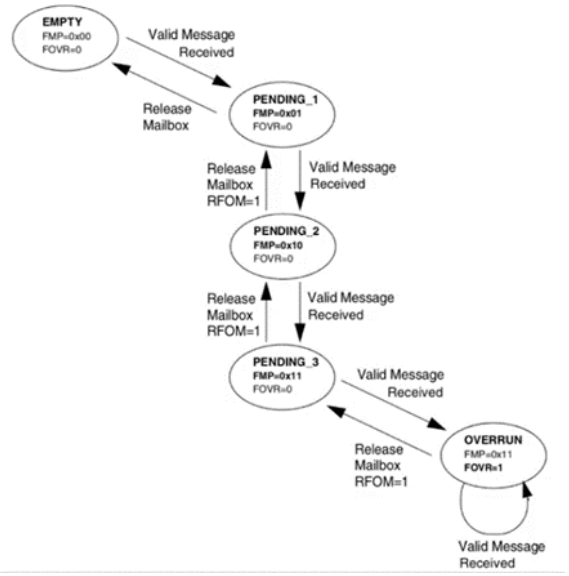

Figure 2.9: Reception Handling

specifies which bits of the identifier should be ignored. This filtering mode is useful when the range of identifiers to be filtered is large and when specific identifiers are not known. On the other hand, in identifier list mode, filter banks are configured to accept messages that match a list of specific identifiers. The filter bank is configured with a list of filter identifiers, and the CAN controller compares the identifier of each incoming message with the list of filter identifiers. This filtering mode is useful when the specific identifiers to be filtered are known.

Chapter 3

Front-end

3.1 Introduction

Qt is a cross-platform application development framework for desktop, embedded and mobile. Supported Platforms include Linux, OS X, Windows, QNX, Android, iOS and others. Qt is not a programming language on its own. It is a framework written in C++. A preprocessor, the MOC (Meta-Object Compiler), is used to extend the C++ language with features like signals and slots . Before the compilation step, the MOC parses the source files written in Qt-extended C++ and generates standard compliant C++ sources from them. Thus the framework itself and applications/libraries using it can be compiled by any standard compliant C++ compiler like Clang, GCC, ICC, MinGW and MSVC. Qt brings its own qmake. It is a cross-platform frontend for platform-native build systems, like GNU Make, Visual Studio and Xcode. Qt comes with its own IDE (Qt Creator). It runs on Linux, OS X and Windows and offers intelligent code completion, syntax highlighting, an integrated help system, debugger and profiler integration and also integration for all major version control systems. GUIs can be written by widgets or QtQuik . In Widgets GUIs can be written directly in C++ . Qt also comes with an interactive graphical tool called Qt Designer which functions as a code generator for Widgets based GUIs. In QtQuick GUIs are written in QML. What is Qt/QML ? QML is a declarative object description language that integrates Javascript for procedural programming. Qt Quick provides the necessary modules for GUI development with

QML. It is possible to write whole applications in QML only, but usually only the GUI is written in QML and the application's backend is implemented in C++ . In our project we first used Qt widgets to write our GUI's application using Qt design studio as it is more easy in use and has many tools and features that helped us to precisely create our design with the specified colors and shapes . It helped us to build the UI , test and iterate it to get the right design we really need . Qt Design Studio provides the designer with the level of control needed to create a very high quality, pixel-perfect UI incorporating rich feedback and animations. It also has plugins for Photoshop, Illustrator, Sketch, Adobe XD, Figma. This plugin let us quickly export all our graphics, components and screens from our design tool as QML components. Then worked with them in Qt Design Studio. With the help of 3D design studio it was easy to create a 3D model , test it and add the required features we wanted to display on it. There was a problem of making a cross compilation of qt design studio on raspberry pi so we had to use Qt Quick to write our GUI's application as it can build high-performance apps rich in visual characteristics and animation with Qt Markup Language. Its features and libraries helped in implementing the required design . Why Qt? With Qt, you can create code that is easy to read, reuse and maintain, which occupies less space and has proven high-quality performance. You'll have the ability to create cutting-edge appealing Qt GUI. Due to its highly productive features, Qt software development takes less time and remains a cost-effective solution. Because of C++ programming language implementation, programmers can use manifold libraries. APIs make Qt app development easier. The framework has cross-platform characteristics. It's easier to create 3D graphical user interfaces with the help of 3D Studio. As a result, your software will have a graphical user interface similar to a native one. The wide choice of modules allows rich functionality in a project.

3.2 Design

3.2.1 First Release

We decided to develop our first design and test it with some inputs and a basic communication protocol after studying and gaining a solid foundation in Qt5/Qml. It comprises of a huge speed gauge in the center and two smaller battery gauges on the sides. It was a terrific first step, but it wasn't the ideal choice, so we began to look for information and gather as much data as we could while keeping up with standards and contemporary shapes. And this leads us to second release

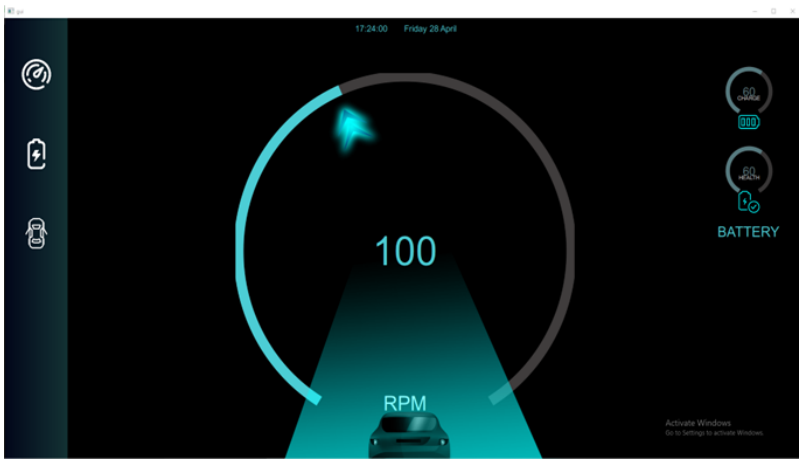

Figure 3.1: first Release gauge

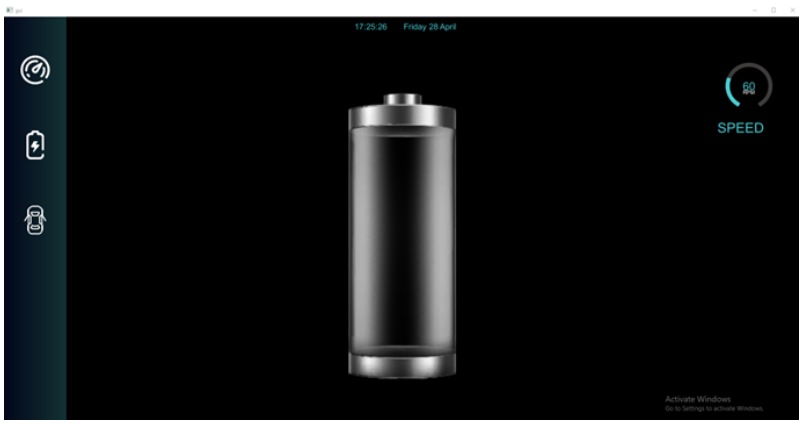

Figure 3.2: first Release battery

3.2.2 Second Release

The two primary gauges are included in the second release. The first one on the right side displays the speed in kilometers per hour, the automobile modes, and the temperature. On the opposite side is the battery gauge (percentage - health - temperature), and in the center is a picture of a car, the date, and the time. The Second Launch is much different from the previous one and is now more standardized. In our opinion, we selected the colors that would be least likely to divert our driver. We did our best to keep it as straightforward and thorough as possible. We made the decision to move on to the next level, therefore we did the third release.

Figure 3.3: second Release

3.2.3 Third Release

We can see that having multiple pages will be a major improvement, so let's start with our home page.

in figure 3.5, we've added a 2D model that is totally dynamic and includes all inputs (doors, lights, tires).

Additionally, the small space in the center can be swiped to display the tire pressure.

For More detailed and informative, larger scale , we consider all pages as in figure 3.6 and 3.7.

The Driver will always be able to view the essential variables clearly so the input, signals and gauges are displayed on every page as well.

In fig 3.7, we offer reliable dynamic models with inputs for both tire pressure and the vehicle.

Last but not least, we began the navigation page by importing a map from many sources as shown in figure 3.8.

We tried many different approaches; the best sources are "osm" and "mapboxgl". Mapboxgl was excellent and offered our customized maps as well, but (Open-Street Map) was just as excellent.

Although we made an effort to completely navigate, there were several challenges, and the lack of a GPS was a major contributor. So, we gave it our best effort and did a nice path route.

Figure 3.4: third Release

Figure 3.5: third Release

Figure 3.6: 2nd Page is Battery Page

Figure 3.7: 3rd Page is car Page

Figure 3.8: 4th Page is Navigation Page

3.3 Implementation

We are working on Qt5/Qml, It has its merits and demerits, as we mentioned before. Let's start with the pages separately.

We just imported our core libraries. Next, we created an item, which may be a rectangle. It is better to declare and define your variables and properties at the beginning and set it to the default. The following code shows example of proprieties defined in the mainpage

```
.
.
.
property bool turnRight:mainWindow.turnRight
property bool turnLeft:mainWindow.turnLeft
property bool seatBelt:mainWindow.seatBelt
property bool lights: mainWindow.lights
property bool chargingState :mainWindow.chargingState
property bool door: mainWindow.door
property bool frontlight:mainWindow.frontlight
property bool backlight:mainWindow.backlight
.
.
.
```

Any element or object should have a width and a height, which is what this anchor is. Therefore, there are two ways to describe the size: first, the width and height; second, anchor it to other borders or elements, which is what we do in this case. so that it takes the parent size, fill: parent. Here, all we need to do is create a file and an object, which we can then use to create more gauges like Battery-Gauge and Speed-Gauge, which serve as our primary gauges.

```
BatteryGauge{
   scale : 0.45
    x:1270
```

```
        y:-40
        socValue: mainPage.socValue
        sohValue: mainPage.sohValue
        sotValue: mainPage.sotValue}
    Speed_Gauge{
      scale: 0.45
      x:-150
      y:-255
      speedValue: mainPage.speedValue
      heatValue: mainPage.heatValue
      smode: mainPage.smode}
```

In Qt, there is a notion called Signal () and Slot (), which simply means that you emit the slot when you receive a signal. In this case, the door state changes, so perform a certain task.
And so forth for each Signal.

```
onDoorChanged:  {
        if(door & !frontlight){
           first.start();
             flag1 = 1;
            flag0=0;
            flag2=0;
            flag3 =0;
          }
        else if(!door & frontlight){
                second.start();
                flag2=1;
                flag0=0;
                flag1=0;
                flag3 =0;
        }
        else if(door & frontlight){
               third.start();
               flag3=1;
            flag0=0;
            flag1=0;
            flag2=0;
```

```
        }
        else{
                zero.start();
                flag0=1;
                flag1=0;
                flag2=0;
                flag3 =0;
        }
      }
```

Here in our main-page as we saw before middle can be both car model and tire pressures, So we made an Elevation of the Car and displayed all tire pressures, We can see the image which is the arrow and the text which is the pressure value ...etc. We will notice that the image source can be either a red arrow when pressure is under set value or green in normal value.

```
ParallelAnimation{
           id:zero
          NumberAnimation{
              target: base
              property: "opacity"
              from:0
              to:1
              duration: transTime
           }


          NumberAnimation{
               target:{flag1 ? doorwithoutL : textt}
               property: "opacity"
               from:1
               to:0
               duration: mainPage.transTime
           }
```

```
        NumberAnimation{
            target:{flag2 ? lwithoutD : textt}
            property: "opacity"
            from:1
            to:0
            duration: mainPage.transTime
             }

        NumberAnimation{
            target:{flag3 ? doorwithL : textt}
            property: "opacity"
            from:1
            to:0
            duration: mainPage.transTime
             }
         }
```

With a wide range of animations, the car model in the center shows the status of the car doors and lights. We can complete all animations simultaneously by using parallel animation. Numerous number animations have been done, as seen above.

```
  Timer{
    id:turnTimer
    interval: 500
    running: mainPage.turnRight || mainPage.turnLeft
    repeat: true
    onTriggered: flasher = !flasher
}

Image {
    id: rightSignal
    x:parent.width-parent.width/3
    y:parent.height/5
    width: parent.width/30
    visible:mainPage.turnRight && flasher
    fillMode: Image.PreserveAspectFit
    source: "qrc:/img/turn-signal_right.png"
```

```
    }
```

Additionally, there are driving signals like the left and right signals. Establish a flashing signal arrow timer. The rest of the car signals continue, each with little variations.

```
Image {
      id: outerTire1
      width:
      carpage.tire1 < carpage.setPressure ? parent.width/39 :parent.width/34
      height:
      carpage.tire1 < carpage.setPressure ? parent.width/26 :parent.width/28
      anchors{
          bottom: tireRect1.bottom
          bottomMargin: carpage.tire1 < carpage.setPressure ? -30 : -28
          right: tireRect1.left
          rightMargin:10
      }
      rotation: -10
      opacity:1
      source:
      carpage.tire1 < carpage.setPressure ?
      "qrc:/img/redTire.png" :"qrc:/img/greenTire.png"

  }
```

We declare our properties (variables) and link them to our primary properties at the beginning of the car-page. Thus, we can quickly access them from a single location.

A green tire for normal pressure and a red tire for low pressure were placed. The pressure value that we obtain from the backend is what we use to determine the color.

After the chart's lib in Qt5 failed to function, we created our own chart using a variable x-axis with time and a y-axis with battery percentage.

```
    Plugin {
      id: mapboxglPlugin
      name:  "osm"          //with mapboxgl we have no Route

}

  RouteQuery {
      id: routeQuery
  }

  RouteModel {              //available pathes
      id: routeModel
      plugin: mapboxglPlugin
      query: routeQuery
      autoUpdate: false
  }
```

In navigation page we want to start with only the map, We imported the QtLocation and with plugin and map item we displayed our first map.

```
            id: myMap
      anchors.fill: naviFrame
      plugin: mapboxglPlugin
      center: QtPositioning.coordinate(carPositionX, carPositionY)
      zoomLevel: zoom
```

Then, with the RouteQuery and Model we get the available paths, And with the mapItemView we can Route and mark on the map.

```
              Canvas {
      id: canvas
      property real startA:root.start
```

```
property real endA :root.end
property real degree: (root.socValue*root.arcLength)/100
//  property real value: slid1.value

anchors.fill: parent
antialiasing: true

onDegreeChanged: {
    requestPaint();
}

onPaint: {
    var ctx = getContext("2d");

    var x = root.width/2;
    var y = root.height/2;

    var radius = root.size/2 - root.lineWidth
    var startAngle = (Math.PI/180) * startA ;
    var fullAngle = (Math.PI/180) * endA ;        //(270 + 360)
    var progressAngle = (Math.PI/180) * (startA + degree);

    ctx.reset()

    ctx.lineCap = 'square';   //lineEdge
    ctx.lineWidth = root.lineWidth;

    ctx.beginPath();
    ctx.arc(x, y, radius, startAngle, fullAngle);
    ctx.strokeStyle = root.secondaryColor;
    ctx.stroke();

    ctx.beginPath();
    ctx.arc(x, y, radius, startAngle, progressAngle);
    ctx.strokeStyle = root.primaryColor;
    ctx.stroke();
}

Behavior on degree {
```

```
            NumberAnimation {
                duration: root.animationDuration
            }
        }
    }
}
```

We created our gauges using the Canvas, a JS-based that allowed me to create pathways and points. From there, we created an arc and put our gauges into action.

Figure 3.9: Diagnostic Buffer

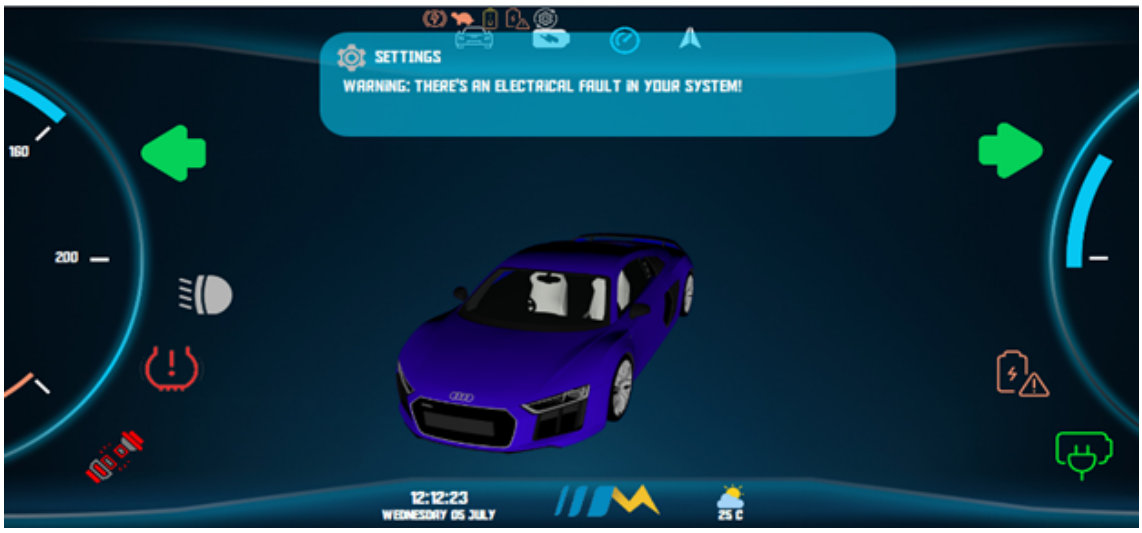

Figure 3.10: Notifications

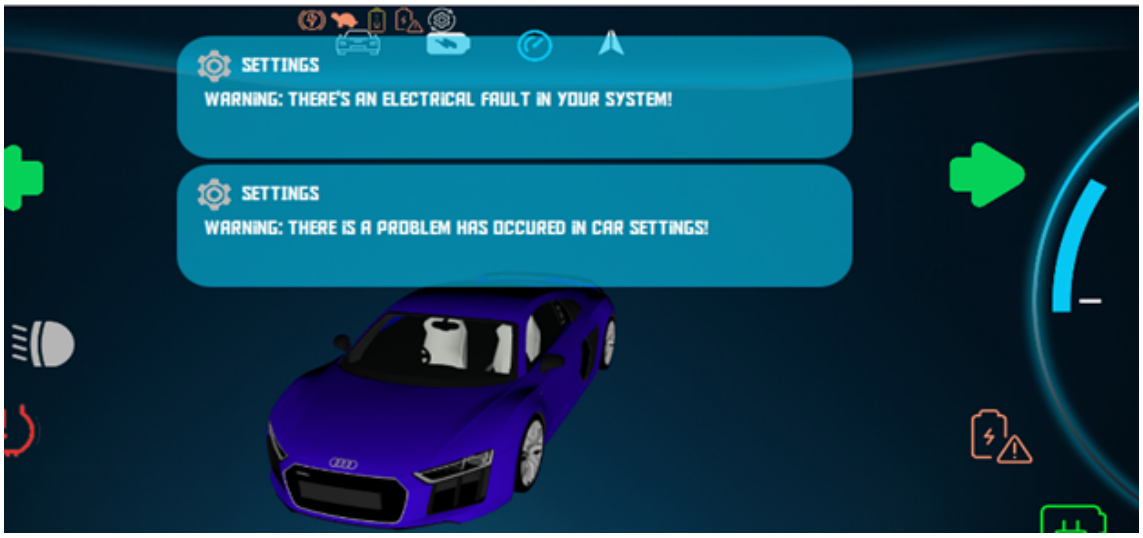

Figure 3.11: every alert sends us notification

3.4 Conclusion

In conclusion, the development of the electric vehicle dashboard project achieved its goal of providing an innovative and user-friendly interface for drivers. Drivers now have access to critical real-time data about their electric vehicles thanks to the introduction of gauges that display speed and battery charge percentage, as well as various pages for complete information presentation. The project's use of two gauges to display the current speed and battery percentage is a simple yet effective way to provide the driver with vital information about the vehicle's performance. Additionally, real-time monitoring of the battery packs' state of charge, state of health, and temperature is crucial in ensuring the battery's longevity and efficiency. The dashboard's notification system alerts the driver in the case of a fault, providing an opportunity to address the issue before it becomes a more significant problem. The system also notifies the driver of any changes in any system, making it easier to identify potential issues and take preventive measures. The diagnostic buffer system is another essential feature of the dashboard, allowing the driver to view the car's history of issues, aiding in the repair procedure. This feature is particularly useful for technicians when servicing the vehicle, as it provides them with valuable information about the car's problems and makes it easier to diagnose and fix issues. The

navigation system with routing provides drivers with a convenient way to plan their route and reach their destination efficiently. The 3D model displaying the car's tires pressure, lights, and door states allows the driver to monitor these critical aspects of the vehicle's performance easily. In summary, the electric vehicle dashboard project is a significant achievement in the field of electric vehicles. The system's ability to provide real-time information about the vehicle's performance, coupled with a notification system and a diagnostic buffer, enhances the safety and reliability of electric vehicles, making them a more attractive option for consumers.

3.5 Future Work

There are several features that could be added to the dashboard in the future to enhance its functionality and usefulness. Here are a few potential ideas:

1. A more complex navigation system: that can deliver real-time traffic updates, helping the driver to avoid traffic and discover the shortest route to their destination. displaying the vehicle's location and providing directions to the driver.

2. A more advanced 3D object: a 3D object that moves smoothly to show the automobile lights, door statuses, and tire pressure.

3. FOTA : can also be used in electric vehicles (EVs) to update the firmware and software that control the vehicle's functions, such as the battery management system, motor control system, and infotainment system. Can also be used to deliver software updates to the vehicle wirelessly, without the need for the owner to bring the vehicle to a dealership or service center for updates.

4. Infotainment system: If the vehicle is equipped with an entertainment system, the dashboard could be used to control it, allowing the driver to play music, videos, or other media.

5. Driver assistance: The dashboard could be designed to provide driver assistance features, such as lane departure warnings, blind spot monitoring, and adaptive cruise control.

6. Personalization: The dashboard could be customized to display the driver's preferred information and settings, such as their favorite radio station, climate control settings, and more.

These are just a few ideas for features that could be added to the dashboard in the future. With advances in technology and the increasing popularity of electric vehicles, there are many possibilities for enhancing the functionality and usefulness of the dashboard.

3.6 Reflection On Learning

Developing an electric vehicle dashboard with the up coming features involves a combination of technical expertise, user-centered design, integration with vehicle systems, and considerations for maintenance and future advancements. It's an exciting endeavor that contributes to enhancing the driving experience and promoting sustainable transportation.

1. Technical Knowledge: Developing an electric vehicle dashboard helped us to have a solid understanding of both hardware and software components and be aware of electronics, sensors, microcontrollers, programming languages, and user interface design which is required to develop it.

2. User-Centric Design: When designing the dashboard we had to be aware of all the users' wants and preferences that must be considered , to ensure that the dashboard is intuitive, simple to use, and presents essential information in an easy-to-understand manner.

3. Integration with Vehicle Systems: To display speed and battery charging percentage accurately, we got the chance to learn how to integrate with various vehicle systems. This may involve retrieving data from speed sensors, battery management systems, or other onboard sensors. Understanding the communication protocols and interfaces used in the vehicle is crucial for successful integration.

4. Information Hierarchy: When displaying diverse information on several pages, it's critical to prioritize and organize data properly. Consider the most important information (speed, battery percentage) and make sure it is always conveniently accessible. Less important information can be shown on additional pages or accessed via menus.

using Qt and QML to implement our dashboard was an excellent opportunity to gain hands-on experience with such a technology as we gained experience in understanding the concept of signals and slots ,dealing with Qt tools , making 3D models with Qt studio, and the ability to improve our experiences.

1. Understanding the Concept of Signals and Slots: Because QML relies on the signals and slots idea in most of its tools, we had the opportunity to comprehend it and use it to obtain all of the car system signals

2. Dealing with Qt Tools: Qt offers a number of tools that make working with it easier, therefore we studied each required tool and all of its features to assist us in implementing our UI.

3. 3- Making 3D Models with Qt Studio: with Qt studio we learned how to deal with a 3D model with its layers , motion and colors , as it made it easier to visualize and design interactive UI elements.

4. 4- Improve Our Experiences: One of the most rewarding aspects of working with Qt and QML is the ability to create visually appealing and interactive user interfaces. QML's design flexibility, along with built-in animations and effects, helped us to improve our experiences.

Chapter 4

Embedded Linux

4.1 Introduction

Embedded Linux refers to the use of the Linux operating system in embedded systems, which are computer systems integrated into devices and machines to perform specific functions. These systems are typically designed to be small, power-efficient, and operate in resource-constrained environments. By utilizing the Linux kernel and associated open-source software, embedded Linux provides a flexible and customizable platform for developing a wide range of embedded applications.

Due to the popularity of Linux operating system, embedded Linux has been introduced to fit the requirements of embedded systems so it has gained significant popularity and has become widely used in various industries due to its numerous advantages.

4.2 Custom Image vs Raspbian Official Image

When it comes to choosing an operating system for the Raspberry Pi, two commonly considered options are Embedded Linux and Raspbian. Here we will compare between these two choices in different terms.

1. **Easy usage and suitability for learning purposes :**
   Raspbian is designed to be user-friendly, making it an excellent choice for beginners or those less experienced with Linux. It comes with a graphical

user interface (GUI) and a familiar desktop environment, making it easier to navigate and interact with the system. In addition, it is optimized for the Raspberry Pi's hardware, ensuring seamless integration and maximum performance. It includes specific drivers and software components tailored to the Raspberry Pi, taking full advantage of its capabilities. These advantages came from the official support from the Raspberry Pi Foundation and the popularity of Raspbian as it is the official recommended operating system for raspberry pi. On the other hand, embedded Linux and image customization needs very good experience in Linux operating systems and needs much more work to do the same job as Raspbian. So in terms of suitability with new learners, Raspbian is more suitable and that's why we have used it at the beginning before learning more and creating our custom image.

2. **Flexibility :**

   - SW customization : Embedded Linux allows users to have greater control over the system configuration. The open-source nature of Linux enables customization and tailoring the operating system to meet specific requirements. Users can choose from a wide range of software components, libraries, and tools to build their applications.
   - HW flexibility : Raspbian is tightly integrated with Raspberry Pi hardware, limiting your options if you decide to switch to a different hardware platform in the future. Embedded Linux, on the other hand, offers support for a wide range of hardware architectures. This flexibility allows you to choose the most suitable hardware for your product's requirements, enabling scalability and adaptability as your product evolves.

3. **Suitability for Products :**

   - Flexibility : The points we have mentioned above makes Embedded Linux more flexible in both SW and HW so we can customize our SW to fit product requirements in addition to the HW target itself we can use any board with limited resources that will be sufficient for our specific product.

- Cost : Embedded Linux eliminates the need for expensive proprietary operating systems, reducing licensing costs for embedded system manufacturers. This cost advantage makes it an attractive choice for organizations aiming to optimize their budgets without compromising functionality or performance.
- Long-Term Support: Commercial embedded Linux distributions often come with long-term support options, providing security updates, bug fixes, and maintenance for an extended period. This is crucial for ensuring the stability and reliability of your product in the market. While Raspbian receives updates from the Raspberry Pi Foundation, commercial embedded Linux distributions may offer more comprehensive and tailored support services.

4.3 Linux Components

Linux image consists of four essential components in order to work and run your applications. figure 4.1 shows 3 of these components interacting together during booting and running conditions. The 4th component which is the toolchain is the first and the most important one which generates all of the three components it's not visible in the figure but it is the reason of the other three existence. In order to create the custom image we will need to generate all of these components. In the following part we will discuss how to generate them.

4.3.1 Methods to Create Linux Image

1. **Roll Your Own (RYO) method :**
this is the method we started with to generate our first image. This method involves manually configuring and building each component from source to create a customized embedded Linux image. This method provides the highest level of control and customization but requires more effort and expertise. Here's an overview of the steps:

- **Toolchain :**

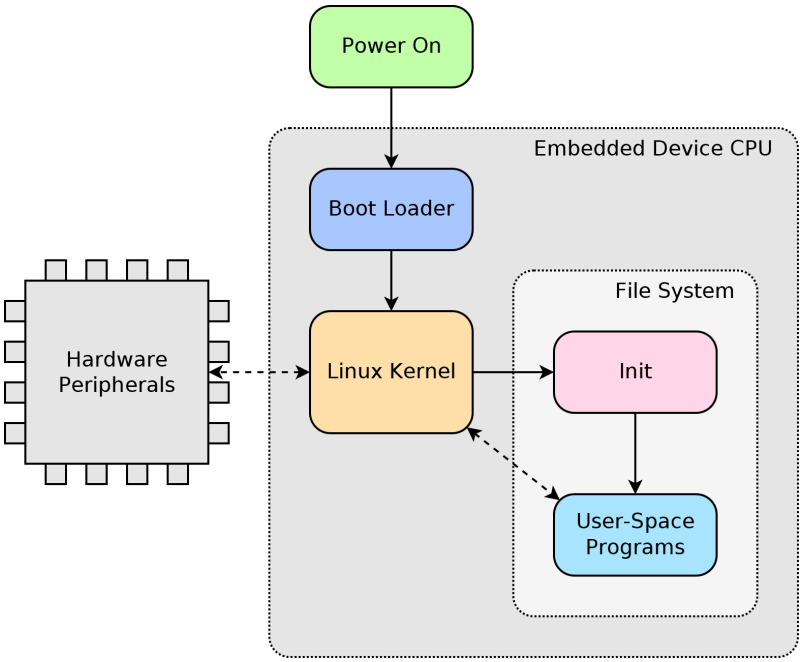

Figure 4.1: Linux components interaction

Each target has its own toolchain used to compile applications and make them ready to run on this target. figure 4.2 shows the components of the toolchain.

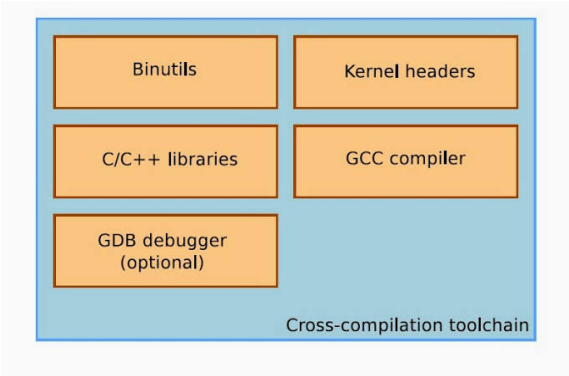

Figure 4.2: Components of Toolchain

In our case we use cross compiler which means the host device and the target don't have the same architecture for example we will use linux OS running on x86 computer to develop our application then we will compile the application and move the resulting executable to run on raspberry pi which has ARM architecture CPU. For toolchain you have 2 options, the first one is to get a ready, suitable, stable and tested one and use it directly or you can create one using a tool called **Crosstool-NG**. You have to choose the one whic accu-

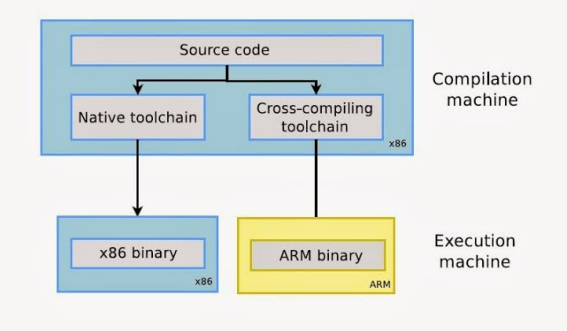

Figure 4.3: Cross Toolchain vs Native Toolchain

rately fits your requirements.We have started to generate our cross toolchain for raspberry pi by following these steps:

– Crosstool-NG installation by cloning repository, installing its dependencies and installing it by using the following commands :

```
$ git clone https://github.com/crosstool-ng/crosstool-ng
$ sudo apt-get install -y gcc g++ gperf bison flex texinfo \
  help2man make libncurses5-dev python3-dev autoconf \
  automake libtool libtool-bin gawk wget bzip2 xz-utils \
  unzip patch libstdc++6 rsync  #dependencies
$ ./bootstrap #setup enviorment
$ ./configure --enable-local #dependency check
$ make
$ make install
```

– Find suitable sample to start configuration on it. This step makes configuration easier as you will start from the default options for your target or something similar. You can find all provided samples by ***list-samples*** argument passed to ct-ng binary but here we will search for our target which is raspberry pi 3. Figure 4.4 shows the output of this command. we will choose one and go on with the following steps.

```
$ ./ct-ng list-samples | grep rpi3
```

– To apply the default configurations use it as an argument to ct-ng.

```
[L...]    aarch64-rpi3-linux-gnu
[L...]    armv8-rpi3-linux-gnueabihf
```

Figure 4.4: List-samples output

```
$ ./ct-ng aarch64-rpi3-linux-gnu
```

– Customization step has come, ct-ng provides simple GUI used for configuration process called ***menuconfig***. Figure 4.5 shows the GUI you can navigate in it and choose any option to edit you can customize anything C libraries, debugging, Architecture, additional libraries and so on.

```
$ ./ct-ng menuconfig
```

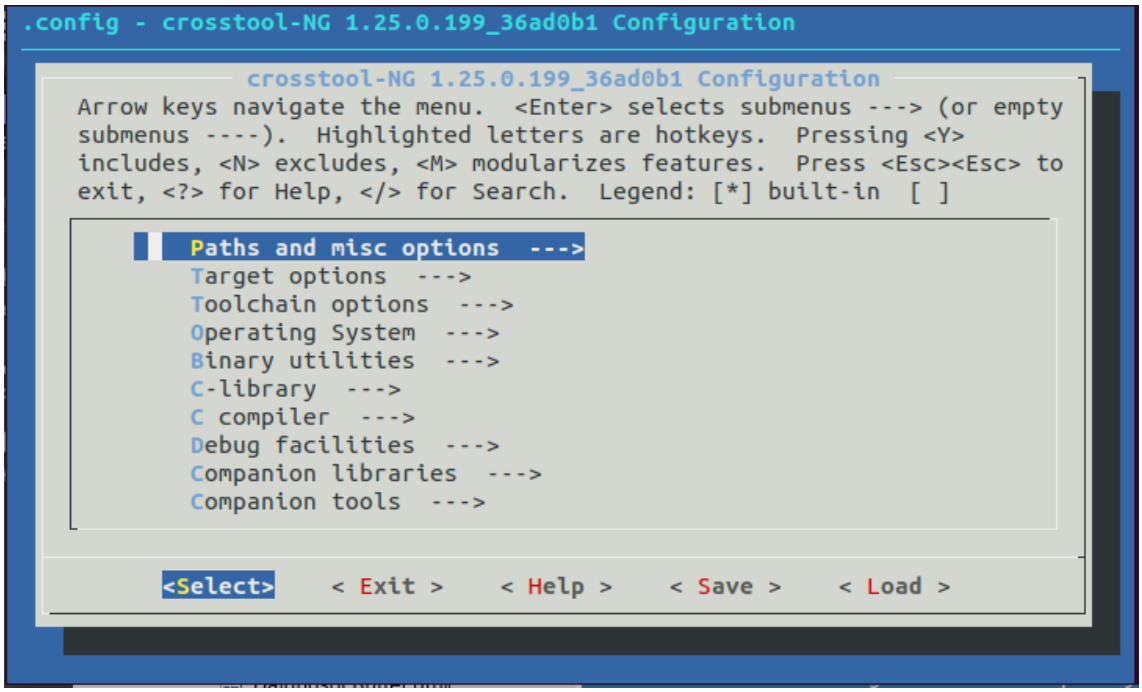

Figure 4.5: Menuconfig of crosstool-ng

This process takes so long and it may cause some problems if you remove some libraries by mistake which may be necessary for some applications so we have decided to use a ready toolchain installed using the following command.

```
$ sudo apt-get install gcc-arm-linux-gnueabihf
```

- **Bootloader:**

  A bootloader is a critical piece of software that initiates the booting process of a computer system. It is typically the first software component to run when a computer is powered on or restarted. The primary function of a bootloader is to load and execute the operating system kernel or other necessary software components into memory, allowing the system to start up and become operational.

  In order to generate custom bootloader a tool called u-boot could be used and here are the steps to use it :

  – u-boot installation by cloning repository, installing its dependencies and installing it by using the following commands :

  ```
  $ git clone https://github.com/u-boot/u-boot.git
  $ sudo apt-get -y install bison flex bc \
    libssl-dev make gcc  #dependencies
  ```

  – Again we will do the same step of finding our target default configurations to start from them. Here you will find the available default configurations in u-boot directory under configs directory. Figure 4.6 shows the available raspberry pi configurations we will choose one and apply it to start configuration.

  ```
  $ ls | grep rpi
  ```

  – To apply the default configurations use it as an argument to make. You have to provide your target architecture and toolchain prefix because they will be needed in the makefile used to apply the default configurations.

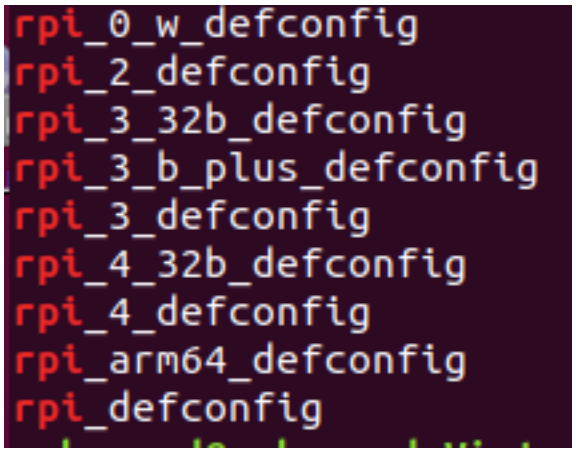

Figure 4.6: u-boot default configs

```
$ make rpi_3_32b_defconfig ARCH=arm \
  CROSS_COMPILE=arm-linux-gnueabihf-
```

– Now we will start configuration using menuconfig too. Here you have a lot of options to customize the rpi boot you can decrease booting time or add some commands to run during booting and many other options.

```
$ make menuconfig ARCH=arm CROSS_COMPILE=arm-linux-gnueabihf-
```

– Build the bootloader using make. The output will be the bootloader binary in different formats but the one we will use is u-boot.bin so we will move it to the sdcard.

The final step to use this bootloader you need to know the boot sequence of the target to know where to insert your u-boot bootloader. In our case we will look at raspberry pi boot sequence which is shown in figure 4.8. In raspberry pi GPU turns on first before CPU so there're some steps using files provided by the manufacturer like bootcode.bin and start.elf and they are not related to CPU. Until we reach the last step where start.elf reads config.txt which contains system configuration parameters and loads the kernel. This file is editable you can edit it to hack the sequence and put your boot loader and then your custom image starts from here. U-Boot provides a command line

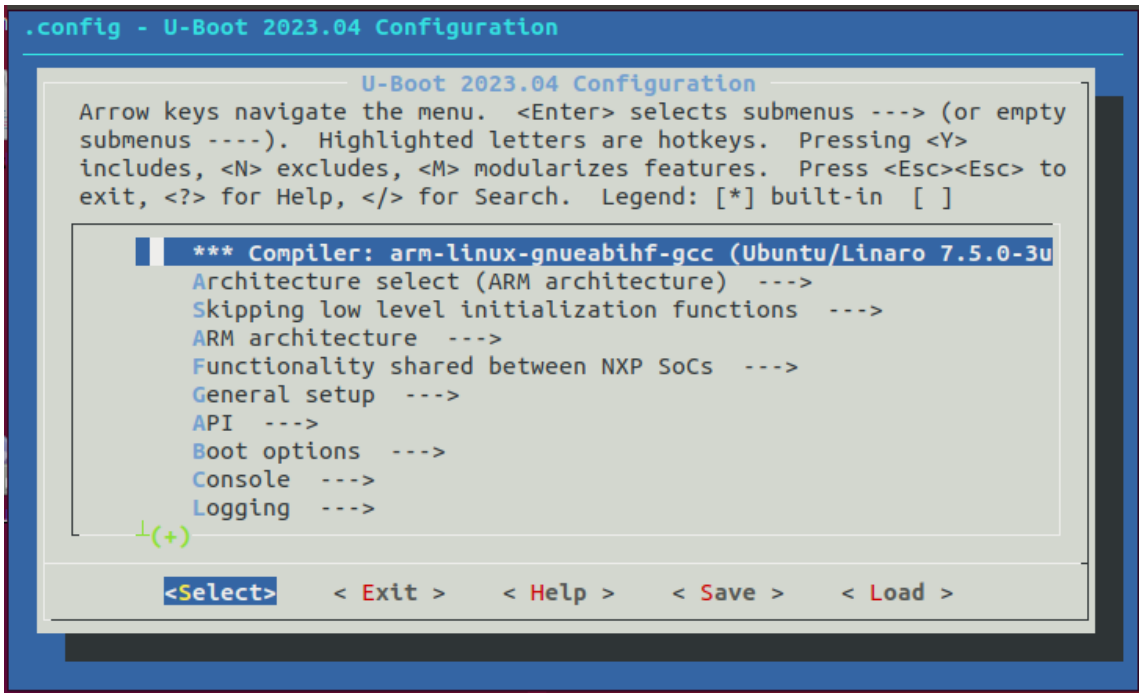

Figure 4.7: Menuconfig of u-boot

interface (CLI) that allows users to interact with and configure the bootloader. Through the CLI, users can set boot options, modify configurations, load and execute images, and perform various debugging and system maintenance tasks.

- **Kernel :**

  Kernel is the core of the operating system which manages everything for you and abstracts user space from HW level. It interfaces with hardware , manages resources and provides APIs used by user space to keep it abstracted. Linux kernel is open source and available on github so many developers took it and customized it to be suitable for specific targets so we will look for kernel version suitable for raspberry pi and customize it to make the process easier.

  – Cloning raspberry pi Linux kernel directory :

  ```
  $ git clone --depth=1 https://github.com/raspberrypi/linux
  ```

  – Again we will do the same step of finding our target default configurations to start from them. Here you will find the available default configurations in Linux directory under arch\arm\configs directory. Figure 4.9 shows the

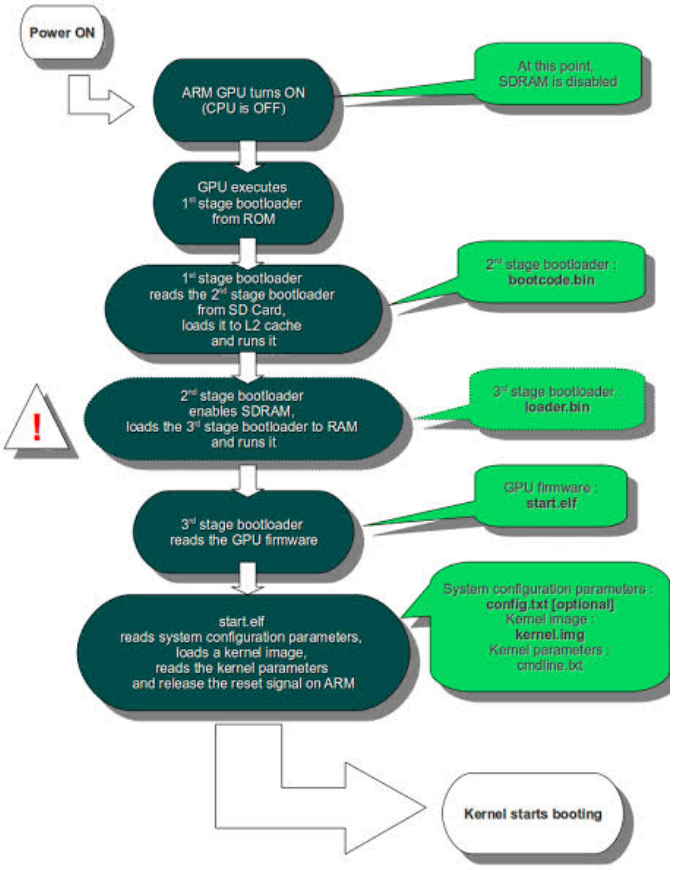

Figure 4.8: raspberry pi boot sequence

available raspberry pi configurations they are defined by bcm here which is the SOC used on raspberry pi. we will choose one and apply it to start configuration.

```
$ ls | grep bcm
```

– To apply the default configurations use it as an argument to make. You have to provide your target architecture and toolchain prefix because they will be needed in the makefile used to apply the default configurations.

```
$ make bcm2835_defconfig ARCH=arm \
  CROSS_COMPILE=arm-linux-gnueabihf-
```

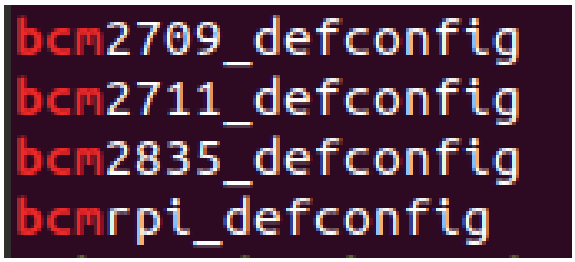

Figure 4.9: kernel available default configs

- Now we will start configuration using menuconfig again. Here you have a lot of options to customize the rpi kernel you can edit kernel modules, device drivers, network drivers, file system support and many other options.

```
$ make menuconfig ARCH=arm CROSS_COMPILE=arm-linux-gnueabihf-
```

- Build the kernel, kernel modules and device tree blobs using make. The output will be zImage file which is the kernel compressed format, dtb which is device tree blob and dtbo device tree blobs overlays.
- Now we will move them as following to Sdcard:
  - zImage ⟶ boot
  - dtb ⟶ boot
  - dtbo ⟶ boot\overlays
  - libs ⟶ rootfs

```
$ make -j12 zImage modules dtbs ARCH=arm \
  CROSS_COMPILE=arm-linux-gnueabihf-
```

- **Root File System:**

  The root file system is a fundamental component of a Unix-like operating system, including Linux. It serves as the starting point and the top-level directory hierarchy for the entire file system. The root file system contains essential direc-

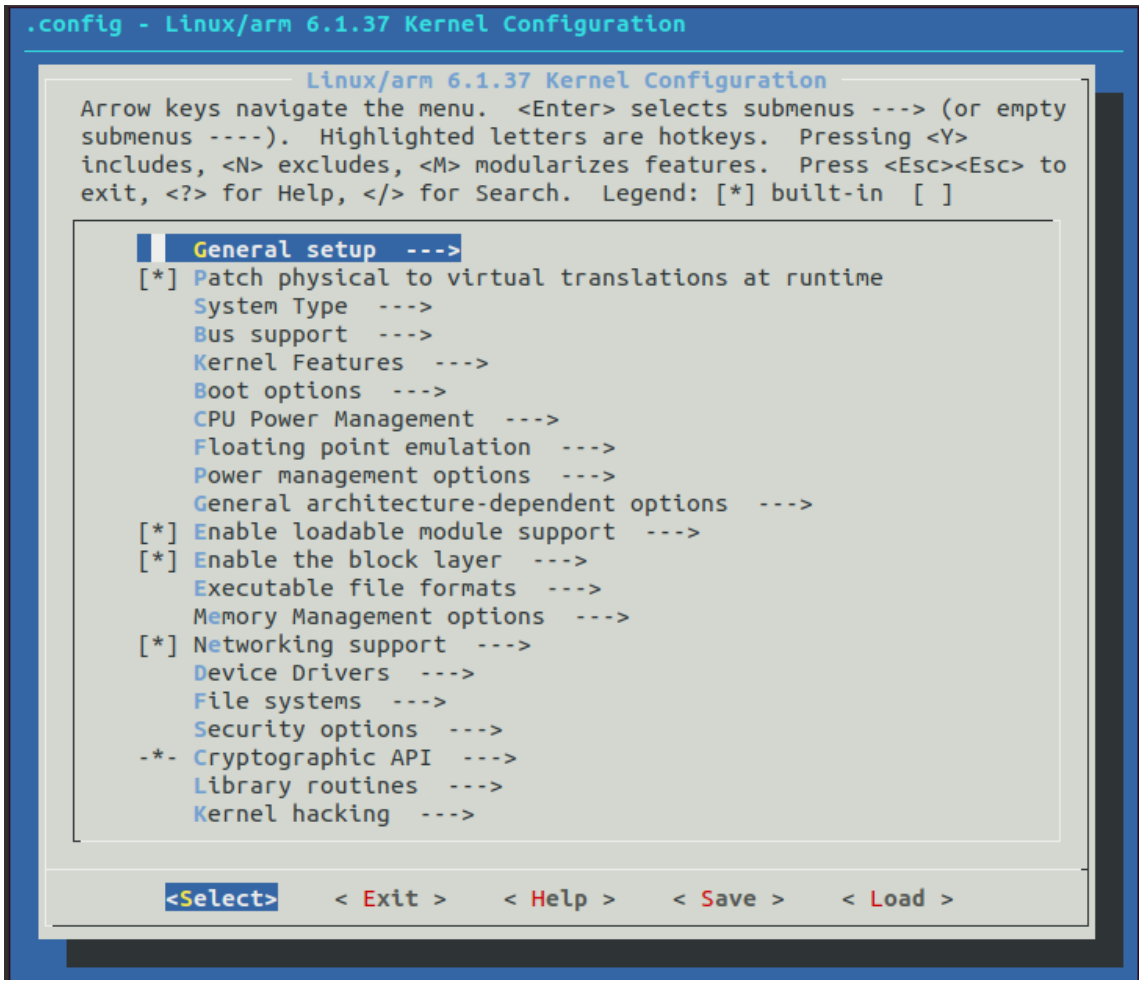

Figure 4.10: Menuconfig of kernel

tories, configuration files, executables, libraries, and other resources necessary for the operating system to function.

The root file system typically includes important directories, such as:

- /bin: Programs essential for all users.
- /dev: Device nodes and other special files.
- /etc: System configuration files.
- /lib: Essential shared libraries, for example, those that make up the C-library.
- /proc: The proc filesystem.
- /sbin: Programs essential to the system administrator.
- /sys: The sysfs filesystem.
- /tmp: A place to put temporary or volatile files

– /usr: Additional programs, libraries, and system administrator utilities, in the directories /usr/bin, /usr/lib and /usr/sbin, respectively.

– /var: A hierarchy of files and directories that may be modified at runtime, for example, log messages, some of which must be retained after boot.

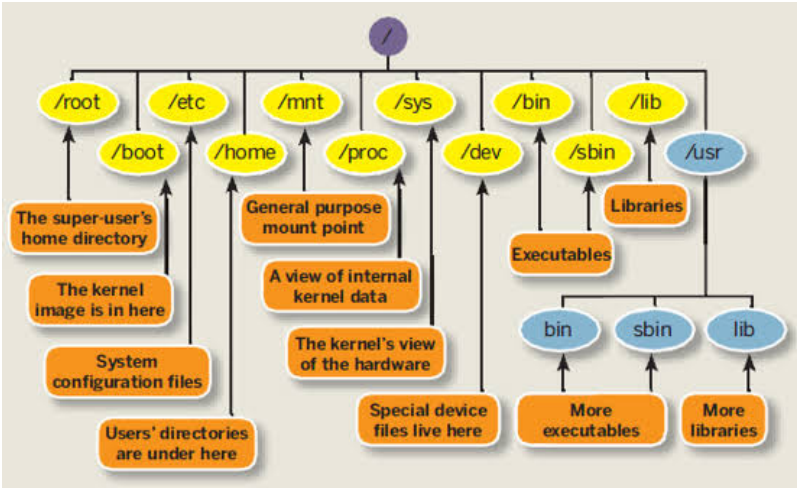

Figure 4.11: Linux Rootfs hierarchy

We create these directories in a staging directory and fill them with the necessary files needed for our minimal image by following these steps:

– Create basic applications binaries needed for shell and other basic tasks. We will use Busybox tool to generate them. As we're talking about minimal image busybox will be excellent choice because of its minimal footprint as it combines all binaries in one binary so it does same functionalities with minimum size and power consumption. We have menuconfig here too you can customize your binaries to be static or dynamic linked, cross compiler prefix, installation path and many other things. We willl move the output to /bin, /sbin, /usr/bin and usr/sbin.

```
$ git clone git://busybox.net/busybox.git --branch=1_33_0 --depth=1
$ cd BusyBox
$ make menuconfig
$ make -j12
```

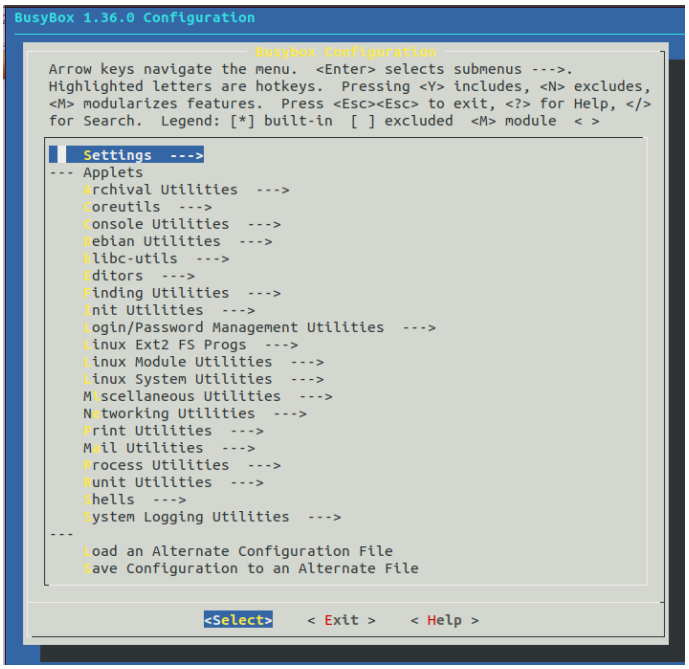

Figure 4.12: Busybox menuconfig

– Create rcS which refers to the initialization script responsible for running essential system startup tasks during the boot process. The "rcS" script is located in the "/etc" directory, specifically "/etc/rcS.d" or "/etc/init.d/rcS". This script is executed by the init system, usually during the early stages of the boot process, before other services and daemons are started.

```
$ mkdir etc/init.d          #creating init.d directory
$ touch etc/init.d/rcS      #creating rcS file
$ chmod +x etc/init.d/rcS #making it executable
```

In the commands above, we have created the file. Next we will fill it by some essential commands needed to be executed at starting.

```
#!/bin/sh
mount -t proc none /proc
mount -t sysfs none /sys

echo /sbin/mdev > /proc/sys/kernel/hotplug
mdev -s  # -s        Scan /sys and populate /dev\n"
```

- Moving libraries to /lib.

Now file system is ready to provide us the minimal functions. We will move all of these directories to the Sdcard rootfs partition.

Finally, we have everything ready to work u-boot.bin, zImage and dtbs in boot partition in addition to the files needed to start GPU provided by raspberry pi manufacturers and rootfs in its partition too. If you edit config.txt file to load u-boot instead of kernel and then use u-boot commands to load zImage and dtbs, then your kernel should start correctly and boot to the basic shell where you can use the commands. To create an application you have to develop it, cross compile it using the toolchain, build its essential libraries statically or dynamically and build any other dependencies like some other applications or device drivers then move all of these to their specific directory in the rootfs. This process takes so long and has many challenges as everything is done manually.

2. **Build Systems:**

Build systems are frameworks or tools designed to simplify the process of building, configuring, and managing embedded Linux systems. These systems provide automation, standardization, and reproducibility, making it easier to create customized Linux distributions tailored for specific hardware platforms. Popular build systems in the embedded Linux domain include Yocto Project/OpenEmbedded, Buildroot, and PTXdist. These build systems offer a structured approach to configure and compile the Linux kernel, bootloader, root file system, and associated software packages. They provide recipe files, configuration options, and a dependency management system to ensure the correct components are built and integrated into the final system image. We have tried both buildroot and yocto, figures 4.13 and 4.14 show the terminal of the two systems. We have decided to work with yocto because of its wide popularity, higher flexibility, layerd architecture and many other reasons makes it more professional and suitable for our application.

Figure 4.13: Buildroot image

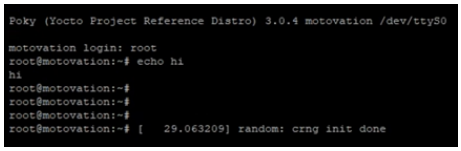

Figure 4.14: Yocto image

4.3.2 Yocto

The primary goal of Yocto is to provide a standardized and reproducible process for building embedded Linux systems. It allows developers to have full control over the software stack, from the Linux kernel to the user-space applications, enabling them to optimize the system's performance, functionality, and footprint. While talking about yocto, you must deal with two other things openEmbedded and poky. OpenEmbedded is the core build automation framework used by the Yocto Project, while Poky is the reference Linux distribution that serves as a starting point for customization within the Yocto Project ecosystem. Yocto project works in layered

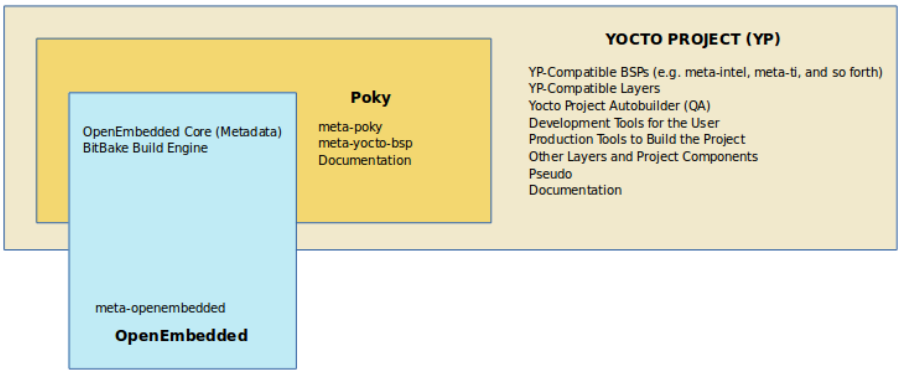

Figure 4.15: Yocto, Poky and OpenEmbedded

architecture. A recipe, which offers instructions on how to find, create, and install a single piece of software, serves as the foundation of Yocto systems. Layers are rationally created to divide up the recipes. Introducing new features, software categories, or hardware they support can be used to arrange recipes in layers. Bitbake is the machine which takes all recipes, configuration files and all data then processes them to generate the output whatever it was an image, SDK or anything else. Figure 4.16 shows the basic layered architecture for raspberry pi minimal image without any

apps running on it. As we have said before yocto is the main umbrella while poky

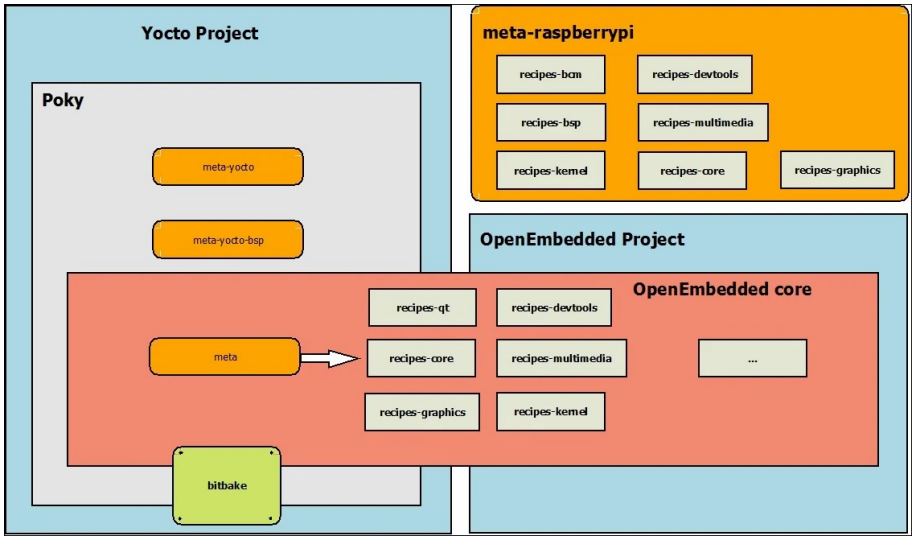

Figure 4.16: Yocto raspberry pi layers

is just an example distribution. OpenEmbedded is a seprate project but it shares metadata with yocto throw meta directory. Meta-raspberrypi is used here because raspberry pi is not one of the supported hardware examples provided by poky so we will need meta-raspberrypi to define raspberry pi machine configurations.

To start working on yocto first we have to run a script in the working directory to initialize environment. Figure 4.17 shows what this script actually does to allow you use bitbake and to let recipes get some details about the working directory. In addition to environment variables, this script creates build directory and creates two configuration files. Local.conf which is the main configuration file read by bitbake and the second one is bblayers.conf which is used to specify layers included and read by bitbake too. Now we will discuss some options nd configurations could be applied in these two files.

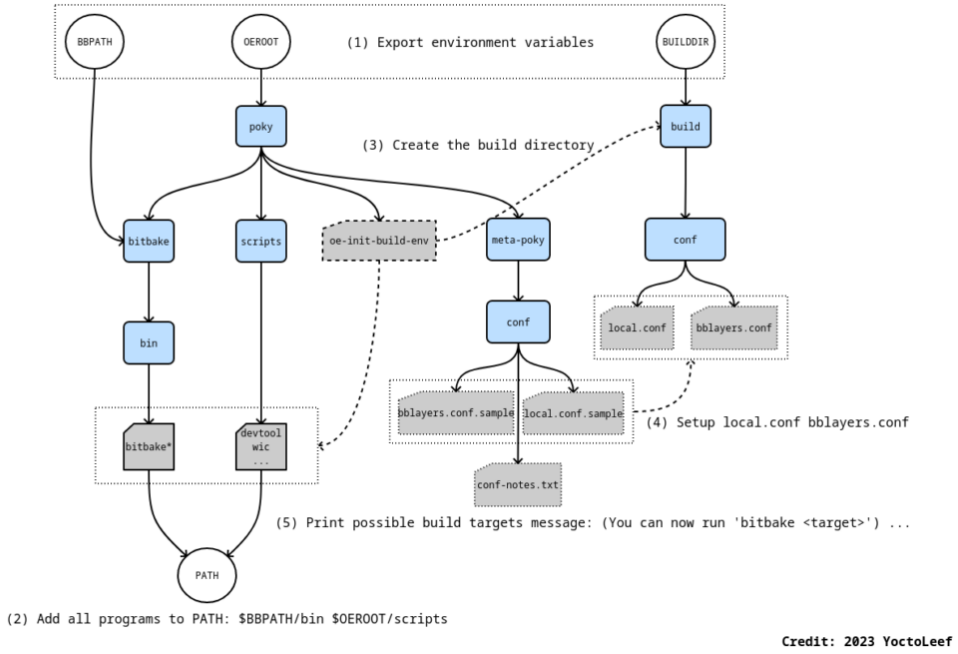

Figure 4.17: How does oe-init-build-env script work

- bblayers.conf :

  you can see all the layers included in this file. If you want to add layer you can add it manually here or just use bitbake-layers script which has a command called add-layer and it does the same thing as adding the path manually.

```
BBPATH = "${TOPDIR}"
BBFILES ?= ""

BBLAYERS ?= " \
  /home/mohammed/yocto/poky/meta \
  /home/mohammed/yocto/poky/meta-poky \
  /home/mohammed/yocto/poky/meta-yocto-bsp \
  /home/mohammed/yocto/meta-raspberrypi \
  /home/mohammed/yocto/meta-openembedded/meta-oe \
  /home/mohammed/yocto/meta-openembedded/meta-python \
  /home/mohammed/yocto/meta-openembedded/meta-multimedia \
  /home/mohammed/yocto/meta-openembedded/meta-networking \
  /home/mohammed/yocto/meta-qt5 \
  /home/mohammed/yocto/meta-mylayer "
```

- Local.conf : In this file there're a lot of options to be set, some of them are shown in the next code :

```
MACHINE="raspberrypi3" #choose rpi3 to be the target machine
INHERIT += "rm_work"   #don't keep files generated during process
ENABLE_UART="1"        #Enables uart in rpi config file
IMAGE_INSTALL_append=" psplash" #add psplash package to be installed
```

4.3.3 Qt Integration :

In order to allow Qt application installation we needed to add Qt layer which will helps in Qt libraries installation and will also generate its toolchain used in qtcreator to build the application. So you may notice that we have included meta-qt5 layer in bblayers.conf file.

Qt Toolchain :

Qt cross-toolchain has to be included in qtcreator so it can generate the executable ready to run on raspberry pi. In order to generate cross-toolchain meta-qt5 provides recipe to build it using bitbake by the following commands:

```
$ bitbake meta-toolchain-qt5
$ cd tmp/deploy/sdk
$ ./poky-glibc-x86_64-meta-toolchain-qt5-aarch64-raspberrypi3-64-toolchain-3.0.2.sh
```

Then we add its path to qtcreator so it can use it to compile the app.

Image :

Images is a special type of recipes used to create Linux image. You can specify many options and packages to be installed on this image. For example our image recipe contains list of qt libraries needed to be installed on raspberry pi and used on our application as shown in figure 4.18. Our image is based on a basic minimal image defined in raspberry pi layer.

```
# Pulled from a mix of different images
require /home/mohammed/yocto/meta-raspberrypi/recipes-core/images/rpi-basic-image.bb
# This image is a little more full featured, and includes wifi
# support, provided you have a raspberrypi3

SUMMARY = "The minimal image that can run Qt5 applications"
LICENSE = "MIT"
MY_TOOLS = " \
    qtbase \
    qtbase-dev \
    qtbase-mkspecs \
    qtbase-plugins \
    qtbase-tools \
"
MY_PKGS = " \
    qt3d \
    qt3d-dev \
    qt3d-mkspecs \
    qtcharts \
    qtcharts-dev \
    qtcharts-mkspecs \
    qtconnectivity-dev \
    qtconnectivity-mkspecs \
    qtquickcontrols2 \
    qtquickcontrols2-dev \
    qtquickcontrols2-mkspecs \
    qtdeclarative \
    qtdeclarative-dev \
    qtdeclarative-mkspecs \
    qtgraphicaleffects \
    qtlocation \
    qtserialbus \
    qtgraphicaleffects-dev \
    qtmultimedia \
"
MY_FEATURES = " \
    linux-firmware-bcm43430 \
    bluez5 \
    i2c-tools \
    python-smbus \
    bridge-utils \
    hostapd \
    dhcp-server \
    iptables \
    wpa-supplicant \
```

Figure 4.18: our Image recipe

Custom Layer :

To use our image and bitbake it we've created a new layer with the image inside it. Figure 4.19 shows the architecture of our new layer with basic example and qt5-my-image which is our custom image.

```
.
├── conf
│   └── layer.conf
├── COPYING.MIT
├── README
├── recipes-core
│   └── image
│       └── qt5-my-image.bb
├── recipes-example
│   └── example
│       └── basicquick_0.1.bb
└── recipes-ext
    └── qt
        └── qtbase_git.bbappend
```

Figure 4.19: our layer architecture

4.3.4 Splash Screen :

Splash screen is by default done by psplash package but we have to edit it to add our logo instead of the default one. There are two option to do this, the first one is to edit its recipe directly but this is not recommended. The solution we have used is to override the recipe using .bbappend file which overrides the default settings without affecting the recipe itself.

4.4 Problems and solutions

- We didn't find canutils in packages available in yocto so we have copied them from the raspberry pi which runs raspbian and added them under /bin .
- We've found while trying the application that some dependencies like libraries and plugins are not installed so we've added them to image recipe and rebuilt the image again.
- Application needs to start automatically on booting so we've created an autostart script under /etc which runs directly after boot then this script executes another script which initializes can bus , starts candump to recieve messages and then starts the application automatically.

```
#!/bin/sh
modprobe can
modprobe can_raw
ip link set can0 up type can bitrate 250000
candump any -n 3
/home/root/new -platform eglfs
```

Chapter 5

Conclusion

5.1 Integration with BMS

The system is very easy to be integrated with any other system communicating in CAN protocol. To prove this concept we have integrated our system with other team BMS system by connecting their CAN-HIGH and CAN-LOW on the CAN bus. The integration is done successfully, we have received messages from their system and processed them to extract data suitable for our display system. Figures 5.1, 5.2 and 5.3 show CAN messages database we've used to extract data from BMS messages.

	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
Byte 0	Error code					Status code		
Byte 1	Battery amp-hours remaining, 0.1Ah resolution, high byte							
Byte 2	Battery amp-hours remaining, 0.1Ah resolution, low byte							
Byte 3	Battery voltage, 0.1V resolution, high byte							
Byte 4	Battery voltage, 0.1V resolution, low byte							
Byte 5	Auxiliary voltage, 0.1V resolution							
Byte 6	Headlights	Isolation integrity (0-100%)						
Byte 7	Temperature (˚C+40)							

Figure 5.1: Message ID 30 description

Value	Name
0	Idle
1	Precharging
2	Running
3	Charging
4	Stopped
5	Setup

Figure 5.2: Byte 1 first 3 bits description

Value	Name	Information
0	No error	
1	Corrupt settings	Invalid settings value detected in memory
2	Overcurrent warning	Current has exceeded the warning threshold
3	Overcurrent shutdown	Current has exceeded fault threshold (drive shut down)
4	Low cell warning	One or more cells below minimum voltage threshold
5	BMS shutdown	Vehicle shutdown due to undervoltage cell for 10+ seconds
6	High cell warning	One or more cells above maximum voltage threshold
7	BMS ended charge	Charger has been stopped due to overvoltage cell for >1sec
8	BMS over-temp	A BMS module has reported a temperature above limit
9	BMS under-temp	A BMS module has reported a temperature below lower limit
10	Low SoC warning	Battery state of charge has passed the low warning level
11	Overtemperature	Temp input on Core has exceeded warning level
12	Isolation error	Insulation fault detected above warning level
13	Low 12V	12V / aux battery voltage below warning level for >5 sec
14	Precharge failed	An error was detected during precharge (failed to start or failed to complete, possibly due to faulty wiring)
15	Contator switch error	Mismatch between contactor state and its auxiliary switch (faulty or seized contactor likely)
16	CAN error	A CAN communications error was detected

Figure 5.3: Byte 1 last 5 bits description

GUI is built to handle most of the messages received in an appropriate way. Battery SOC, SOH, temperature and voltage are calculated and displayed on gauges. Errors, faults and warnings are added to Notifications and diagnostics buffer to notify the user when any fault happens. In addition, there are some indications like battery is charging or not this is also displayed on the screen. There is a separate page used to view everything related to battery. Figure 5.4 shows battery page which contains gauges and some other information related to BMS system.

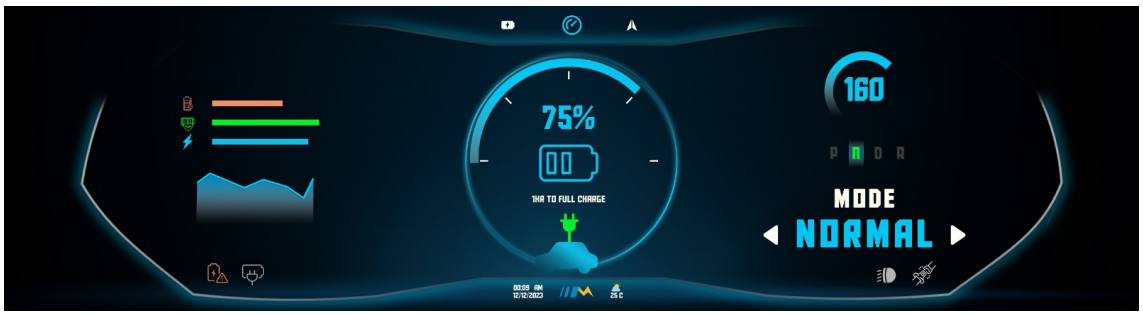

Figure 5.4: battery page design

5.2 Results

We have included some visual representations of the running application on the dashboard screen. These images offer a closer look at the user interface and highlight specific features and functionalities. You will find visuals displaying the battery level and charging status, real-time energy consumption data, intuitive gauges indicating speed and power usage, and the display of range estimation. Additionally, we have captured screenshots depicting the integration with the BMS, illustrating how the dashboard seamlessly collects and presents vital information related to the battery's health and performance. These photos provide a visual demonstration of the application's effectiveness in delivering crucial data to the driver in a clear and user-friendly manner.

Figure 5.5: our custom image booting with our logo

Figure 5.6: example of signals shown on screen : front lights indication

Figure 5.7: battery page taking readings and displaying them

Figure 5.8: diagnostics system showing some error

The successful integration with the BMS serves as a robust indicator of the system's potential for seamless integration with other automotive systems. This achievement highlights its compatibility and adaptability, paving the way for future expansions and integrations. In the realm of automotive applications, precision and dependability are paramount. Thus, it is imperative to ensure that the developed system meets stringent performance and reliability requirements. By prioritizing these aspects during the development process, the digital dashboard project demonstrates its commitment to delivering a high-performing and dependable solution. Moreover, considering the dynamic nature of the automotive industry, it is essential to design the system to be adaptable to potential future changes. This adaptability enables the system to accommodate evolving technologies, industry standards, and emerging requirements without significant disruptions or modifications. By emphasizing the success of the BMS integration, as well as highlighting the importance of meeting performance and reliability criteria, the project underscores its potential for seamless integration into automotive applications. Additionally, the project acknowledges the necessity of creating a system that is both robust and flexible, capable of withstanding future changes and ensuring long-term usability. In conclusion, the accomplishment of integrating with the BMS provides a strong indication of the system's ease of integration with other automotive systems. It also underscores the project's dedication to meeting high accuracy and reliability standards. By prioritizing adaptability and future-proofing, the system can successfully navigate potential changes and ensure continued performance and relevance in the dynamic automotive landscape. The system's inherent flexibility and customizability make it well-suited for adapting to the unique requirements of different car systems. This capability ensures that the digital dashboard can be easily customized and modified to support additional functionalities such as Advanced Driver Assistance Systems (ADAS) or other emerging technologies. The system's architecture and design allow for seamless integration of new features, enabling it to evolve alongside advancements in the automotive industry. Whether it involves incorporating ADAS capabilities or accommodating other innovative technologies, the system is designed to be adaptable and open to customization. By leveraging the system's customizable nature, car manufacturers can tailor the digital dashboard to their specific vehicle models

and integrate it with various advanced functionalities. This flexibility empowers car manufacturers to enhance the user experience and provide cutting-edge features that align with the evolving needs and demands of their customers. In summary, the system's ease of customization and adaptability positions it as a versatile solution capable of supporting various automotive systems. With its inherent flexibility, the digital dashboard can be modified to integrate seamlessly with different functionalities, including ADAS or other emerging technologies. This ensures that the system remains relevant and capable of meeting the changing requirements of modern car systems.

5.3 Future Work

In future development, there is a range of potential features that can be added to the dashboard to enhance its functionality and usefulness. One key area for improvement is the implementation of a more advanced navigation system, providing drivers with enhanced route guidance and real-time traffic updates. Additionally, the dashboard could benefit from a more sophisticated 3D object representation, enabling the incorporation of visually engaging and interactive elements. By exploring these possibilities, the dashboard can evolve to offer a more comprehensive and immersive user experience: 1) Advanced Navigation System: Develop a more sophisticated navigation system that provides real-time traffic updates, offers alternative route suggestions, and includes features like live weather updates and points of interest. 2) Enhanced 3D Objects: Implement smoother and more interactive 3D objects to depict automobile lights, door statuses, tire pressure, and other dynamic information. 3) Firmware Over-the-Air (FOTA) Updates: Integrate FOTA capabilities to enable wireless updates of firmware and software controlling vehicle functions, ensuring the latest features, bug fixes, and security enhancements can be delivered remotely to the car. 4) Infotainment System Integration: Explore integration with the vehicle's entertainment system, enabling control of music, videos, media streaming, and personalized preferences directly from the dashboard. 5) Advanced Driver Assistance Features: Incorporate driver assistance functionalities such as lane departure warnings, blind spot monitoring, adaptive cruise control,

and collision avoidance systems to enhance overall driving safety. 6) Personalization Options: Provide customization options for the dashboard, allowing drivers to personalize the display with preferred information, settings, themes, and personalized profiles. As technology continues to evolve and user preferences change, these potential features can enhance the dashboard's capabilities, further improving the driving experience and adding value to the overall system. Exploring these ideas in future development efforts will contribute to the continuous evolution and innovation of the digital dashboard in electric cars.